ADA266889

TECHNICAL REPORT NO. 532

ARMY SOFTWARE TEST AND EVALUATION PANEL (STEP)

SOFTWARE METRICS INITIATIVES REPORT

HENRY P. BETZ
PATRICK J. O'NEILL

APRIL 1993

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

DTIC QUALITY INSPECTED 6

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification .....
By .....
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

ACKNOWLEDGEMENTS

The U.S. Army Materiel Systems Analysis Activity (AMSAA) recognizes the following individual for contributing to this report:

Peer Reviewer: Barry H. Bramwell, Reliability, Availability, and Maintainability Division (RAMD)

Authors: Henry P. Betz, RAMD, Patrick O'Neill, Combat Support Division.

The next page is blank.

CONTENTS

CONTENTS (Continued)

ARMY SOFTWARE TEST AND EVALUATION PANEL (STEP)
SOFTWARE METRICS INITIATIVES REPORT

1. INTRODUCTION

This report documents the efforts and portrays the findings and recommendations of the Army Software Test and Evaluation Panel (STEP) Subgroup on Measures. The final recommendations include the mandatory use of a minimum, common set of software metrics in order to better measure and manage software development from the test and evaluation perspective. This set of measures should be used to help determine whether the system under consideration has demonstrated the necessary functionality, maturity, and readiness to proceed to the next stage of development or testing. This set of metrics is a minimum set; other metrics of specific interest to a certain agency or point of view can and should be collected if desired.

Of course, there are many characteristics of the software that need to be evaluated for quality and conformance but do not lend themselves to measurement. These quality factors remain important to any evaluation of software. Metrics are simply one element of a set of tools that should be used in a software evaluation.

2. MEMBERSHIP

The Measures Subgroup of the STEP was chaired by the U.S. Army Materiel Systems Analysis Activity (AMSAA). The following organizations participated in the subgroup activities: AMSAA; Test and Evaluation Command (TECOM); Communications and Electronics Command (CECOM) Concurrent Engineering Directorate; CECOM Center for Software Engineering; Armaments, Munitions, and Chemical Command (AMCCOM) Product Assurance and Test Directorate; Information Systems Support Command (ISSC); Information Systems Engineering Command (ISEC); Operational Test and Evaluation Command (OPTEC); Headquarters, Army Materiel Command (HQ, AMC); the Software Engineering Institute (SEI); Air Force Operational Test and Evaluation Center (AFOTEC).

3. OBJECTIVES

The objective of the Measures Subgroup was to force Army level focus on the application of the principles of Total Quality Management (TQM) to the development and management of software. Specifically, the group's mission was to develop a common set of measures which can be used throughout the software life cycle to judge the maturity and readiness of the software to proceed to the next stage of development or testing.

4. METHODOLOGY

The Measures Subgroup was formed because the consensus of the STEP was that measuring software characteristics may help to improve the Army's ability to manage software development. The subgroup did not by itself study the current software development, test, and evaluation process; that effort was undertaken and documented by other STEP subgroups. Hence, the implicit assumption was that the measurement of software characteristics should be applied regardless of the process framework.

The problem of developing a set of measures was approached from the viewpoint that first, it must be determined what was important to measure, and second, appropriate metrics must be found to measure those important characteristics of the software. The emphasis on TQM meant that both process and product measures had to be developed, so that the process can continually be improved.

The group first attempted to capture the state of the Army in software measurement, as well as the state of the art. These states were examined by way of literature searches, limited case studies, and discussions with recognized experts in the academic world, industry, and government.

The group also recognized early on that it was important to have a tractable set of measures. It was imperative that the most important software characteristics be measured, while keeping the sheer volume of metrics to a manageable number. Along the same lines, thought was given to the development of metrics whose inputs would be relatively easy to collect. These goals were important in order to avoid yet another exhaustive compendium of metrics. To be acceptable to both the Army's development community as well as the test and evaluation community, the set of measures must be necessarily tractable while covering the entire spectrum of software characteristics that are important to test and evaluation. The group's concern about having a minimum set of metrics was later reinforced and applauded by NASA personnel, who are recognized in many circles as the government leader in the use of software metrics.

5. METRIC CRITERIA

In order to fit the somewhat restrictive guideline of representing a single, minimum, useful set of measures, it was necessary to establish certain criteria for accepting a metric.

First, the metric had to address the objectives and mission of the group. Therefore, focus was placed upon the following areas of interest: demonstrated results under stress loads, demonstrated functionality, completeness of the development process, and readiness of the product for advancement to the next milestone or test.

Second, in order for the metric to be useful it was felt that it must be unambiguous; possess a prescribed method for data gathering and evaluation; be easy to gather and non-labor intensive to evaluate; be consistently interpreted; prescribe resulting actions that are objective, timely, and finite; and have intrinsic worth (i.e., demonstrate value-added).

Lastly, the group also agreed that, to the extent possible, questionnaires, weighting schemes, and other subjective techniques would be avoided.

6. FINDINGS

The group quickly discovered that there is no shortage of metric ideas. Indeed, many organizations (government and private sector) use some type of software metrics, and many academic researchers are pursuing novel ideas and approaches. The group consensus, as supported by both document reviews and personal contacts, was that the Army does not in general use metrics to help measure software readiness. While there are a few pockets of metric use in the Army, those pockets use metrics in differing fashions. Furthermore, the use of metrics is not approached in an engineering fashion.

A related problem in the Army (as indicated through the case studies performed by the STEP) is that even when metrics are specified in documents like the Test and Evaluation Master Plan (TEMP), the data are often not collected. Further, in some instances the basic data are collected but are not reported in a timely fashion.

Through literature searches, months of subgroup efforts, case studies, and visits to key industrial, government, and academic institutions, the subgroup arrived at a minimum set of software metrics. This minimum set was sent through a limited Army staffing. Section 7 describes the recommended metric set in detail.

7. METRIC SET

The recommended minimum metric set is described in this section. In the course of the subgroup activities, the number of metrics varied between nine and fifteen. The final outcome was a recommended minimum set consisting of twelve metrics. Two of the additional three metrics, while not part of the minimum set, are described in Appendix A as optional. These two metrics (manpower, development progress) were moved out of the minimum set because of several reasons: they were among the most costly metrics in terms of data collection; their basic data are often collected as part of existing program development or test and evaluation activities; the set of fifteen was deemed too large. The metric on documentation, which was in earlier versions of the set, has been deleted entirely, primarily due to the fact that the group considered it too subjective for purposes of measurement, as well as overly labor intensive. An assessment of documentation is typically performed as a routine part of an independent government evaluation.

The metrics are not given in priority order. Rather, we have attempted to group them in a logical fashion: management metrics, requirements metrics, and quality metrics. The management metrics (cost, schedule, computer resource utilization, software engineering environment) deal with management, contracting, and programmatic issues. The requirement metrics (requirements traceability, requirements stability) deal with the specification, translation, and volatility of requirements (reqts). The quality metrics (design stability, complexity, breadth of testing, depth of testing, fault profiles, reliability) deal with testing and quality aspects. Cost and schedule, in addition to being useful as measures, should also be helpful in evaluating the utility of the other metrics (e.g., as predictors).

The metrics span both process and product measures. However, there is not a clean delineation between which metrics are process measures and which are product measures. Indeed, some measures fall into both categories.

Several things must be kept in mind in reading this section. In keeping with the underlying STEP tenet of encouraging a single Army software T&E process, an attempt was made to address both systems governed by the AR 70 series of regulations and those governed by the AR 25 series of regulations. That is, the metrics are intended to address a single Army process for both Materiel Systems Computer Resources (MSCR) (formerly Mission Critical Computer Resources (MCCR)) and Automated Information Systems (AIS).

Many terms are tied to the life cycle model portrayed in DOD-STD-2167A and DOD-STD-2168. However, the entire metrics definition activity, as well as all other STEP activities, has been predicated on the recognition that many Army systems currently being developed are using a non-traditional system acquisition model, including several derivatives of the Army Streamlined Acquisition Program (ASAP). For example, many of today's systems are following an evolutionary acquisition strategy (e.g., the Army Tactical Command and Control System (ATCCS) and its component systems). Some of the systems chosen as case studies for the Metrics Subgroup (e.g., the All Source Analysis System and the Maneuver Control System) are being developed using non-traditional

strategies. It is the firm belief of the STEP that the metrics can also be applied to systems using these non-traditional strategies.

In a similar vein, many terms used to depict the software life cycle are terms which traditionally pertain to a waterfall software life cycle model. The metrics can also be applied to systems whose software is being developed following either a "spiral" life cycle model or some other non-traditional derivative.

While some of the metric descriptions assume that the software is being developed by a contractor, an attempt has been made to use the term "developer" in lieu of "contractor." In this context, the term "developer" refers to either a private contractor or an in-house government development activity. Regardless of who the developer is, the same metrics should be used.

It must also be pointed out that the graphs shown in this section are merely for illustration. Many trends are possible for each metric. Additionally, there are other ways of processing and displaying the data to be collected that may be very appropriate for specific systems.

Finally, it must be stated that the metrics should be used as indicators; they should be used to portray trends over time, rather than placing too much importance on a calculated value at a single point in time. The trends can be studied in and of themselves, or they can be compared with trends from similar systems that have already been built. Within the body of this section, we have tried to specify rules of thumb for those metrics where experience seems to indicate that a single value makes sense. As the metrics become proven, validated entities, numerical thresholds and exit criteria may emerge. Even as a scientifically validated set, however, the metrics remain most useful when used as indicators of trends.

Table 7-1 presents a list of the metrics with a summary of the primary characteristics the metrics are intended to address.

Table 7-1

Metric Category	Metric Name	Objective	Measurement
Management	Cost	track s/w expenditures	$ spent vs $ allocated
	Schedule	track schedule adherence	milestone slippage
	Computer Resource Utilization	track planned/actual resource usage	% utilization
	S/W Engineering Environment	rate developer s/w engineering environment	maturity level
Reqts	Reqts Traceability	track reqts down to code	% reqts traced
	Reqts Stability	track changes to reqts	# reqts changes
Quality	Design Stability	track design changes	stability index
	Complexity	assess code quality	complexity indices
	Breadth of Testing	track testing of reqts	% of reqts tested % of reqts passed
	Depth of Testing	track testing of code	% of paths tested, etc.
	Fault Profiles	track open vs closed anomalies	# of faults average open age
	Reliability	assess s/w mission failures measure down time	MTBF restoration times

7.1 Metric: Cost.

Purpose/Description:

The cost metric provides insights into how well the cost of software development is being controlled. The metric takes advantage of existing acquisition policies with an emphasis added to ensure that software items and work are properly identified and allocated. It provides visibility of current cost and overall schedule variances, indications of likely trouble areas, trends for future costs and overall schedule, and estimated cost at completion of the development effort.

Additionally, historical data on this metric will be one of the primary measures of program success or failure. An attempt will be made to correlate cost (in conjunction with other measures of success/failure (e.g., schedule, degree of user acceptance)) with the relative predictive value of other metrics, thus aiding in the validation process.

Life Cycle Application:

Planning for cost data collection begins in the development of a Work Breakdown Structure (WBS) for each Request For Proposal (RFP), ensuring that software items are incorporated into the WBS and retained in the Contract WBS (CWBS). This metric continues through the life of the contract(s). If the software developer is a government agency, the same cost data must be collected.

Algorithm/Graphical Display:

Cost data are collected in accordance with current Department of Defense (DOD) and Department of the Army (DA) policies for Cost/Schedule Control Systems Criteria (C/SCSC), Cost/Schedule Status Report (C/SSR) or Contract Funds Status Report (CFSR), whichever is appropriate for the scope of the overall program effort. Software costs and schedules must be readily identifiable in the reports. This could be accomplished with a WBS that separates software from other aspects of the program, or through a simple data base management system that extracts the software metrics data from the larger data set provided by the developer.

Figures 7.1-1 and 7.1-2 show typical cost/schedule displays. Figure 7.1-1 displays cumulative cost and schedule data using "earned value" and actual costs relative to that which is planned, while Figure 7.1-2 displays similar information as deviations from the contract cost/schedule. They can be cumulative for the whole program or broken out for critical or high risk area displays.

The following definitions should be used in implementing the cost metric:

Budgeted Cost of Work Scheduled (BCWS) - the sum of the budgets for all work packages, the level of effort, and apportioned effort scheduled to be accomplished within a given time period.

Budgeted Cost of Work Performed (BCWP) - the sum of the budgets for completed work packages and completed portions of open work packages, plus the applicable portions of the budgets for level of effort and apportioned effort. BCWP is also called earned value.

Actual Cost of Work Performed (ACWP) - the cost actually incurred in accomplishing the work performed within the given time period.

Contract budget baseline - the total of the original contract target cost plus negotiated contract changes plus the estimate cost of all authorized, unpriced work.

Estimated Cost at Completion (EAC) - the sum of all actual costs to date plus the estimate for work remaining.

Management Reserve (MR) - that portion of the budget withheld for management control purposes rather than designated for the accomplishment of a specific task.

In addition to the computation of these basic parameters, the following values should also be computed:

Cost variance = BCWP - ACWP

Schedule variance = BCWP - BCWS

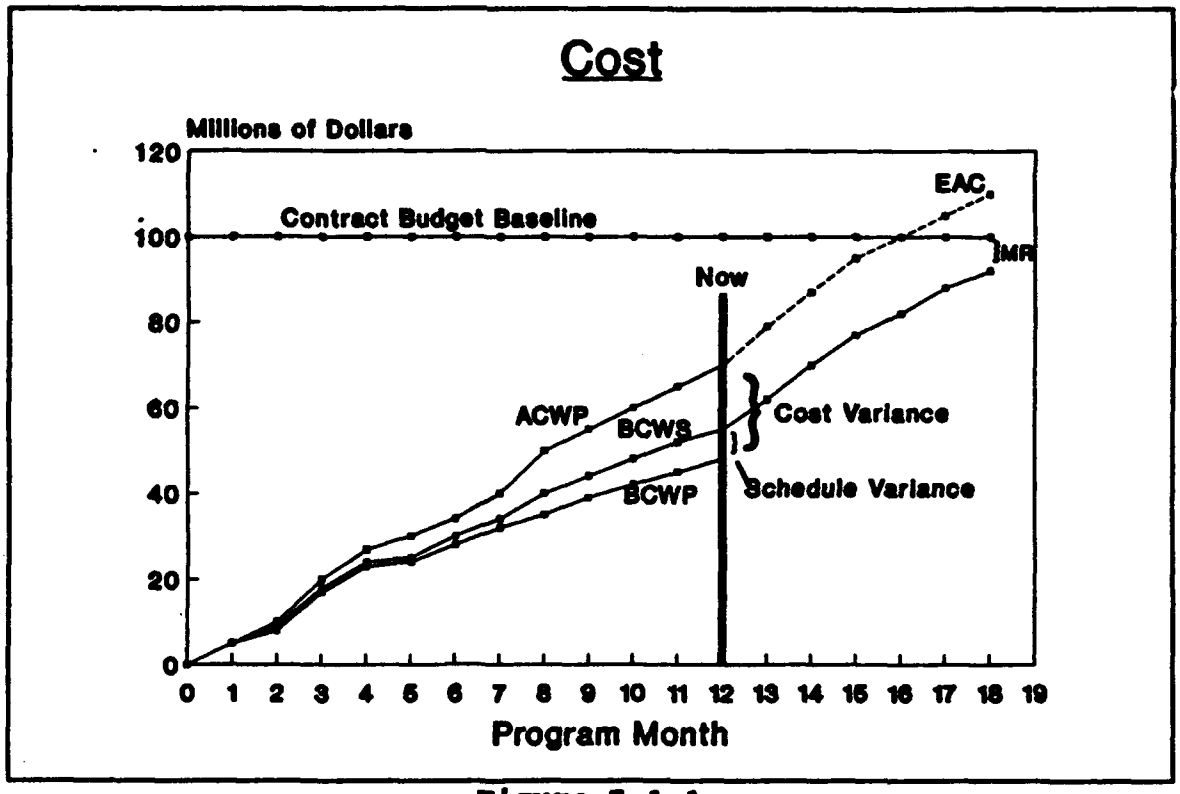

Figure 7.1-1

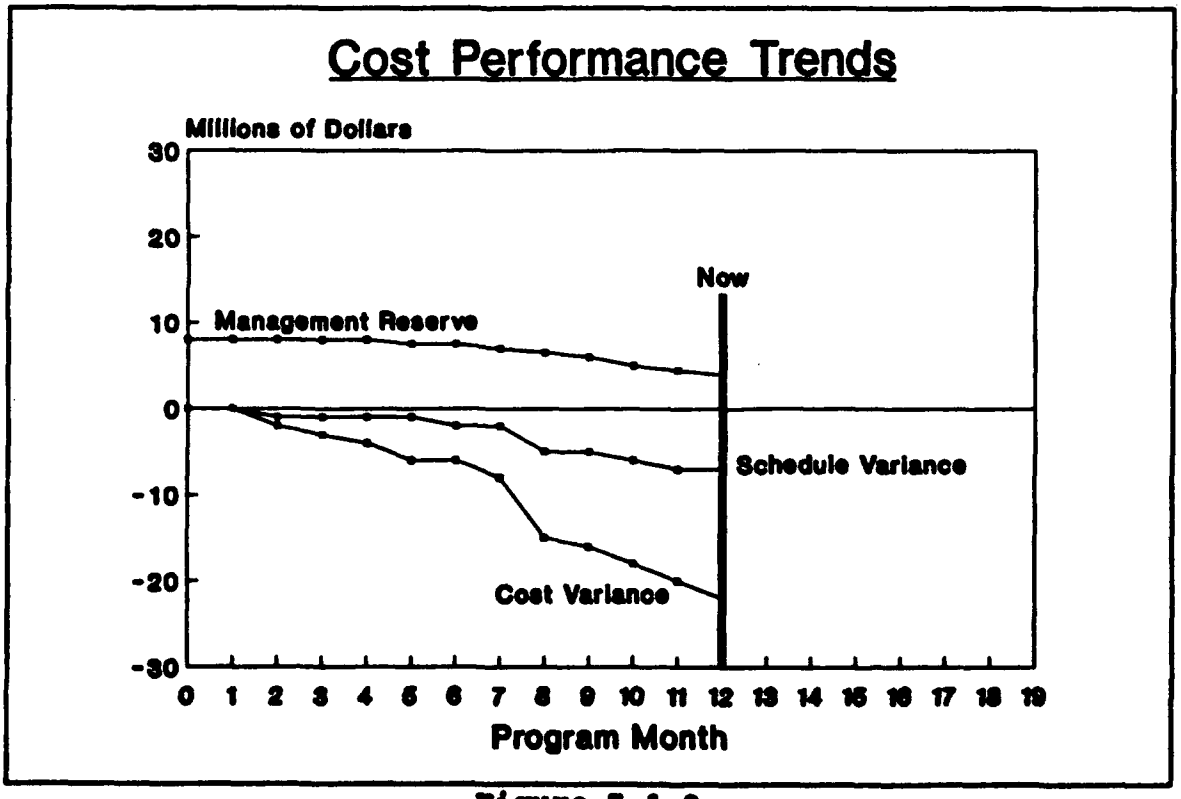

Figure 7.1-2

Data Requirements:

Cost and schedule data must be collected at least to the level of the CWBS which provides the following data. Data from lower levels are optional; reporting is based on the degree of risk that a particular configuration item presents to the overall program. MIL-HDBK-WBS.SW shows how to configure a good CWBS that will cover the majority of software costs associated with a program. Some tailoring may be required to address program peculiarities.

For purposed of the data requirements, the following activity types are defined:

Software activity types (for total project):
- CSCI(s) integration and testing
- verification and validation
- software project management
- software engineering management
- software quality assurance
- software configuration management

Software activity types (for each CSCI):
- requirements analysis
- design
- coding and unit testing
- CSC(s) integration and testing
- FQT
- problem resolution
  - investigation
  - redesign
  - recoding and unit testing
  - CSC(s) re-integration and testing

As a minimum, for C/SCSC and C/SSR qualifying systems, the following data should be supplied for each software activity type listed above:

BCWS
BCWP
ACWP
Schedule Variance
Cost Variance

In addition, the following data should be collected for the total project:

BCWS
BCWP
ACWP
Schedule Variance
Cost Variance

EAC
Management Reserve
$ invested in tools
$ invested in new equipment and facilities

Equipment, facilities, and tools shall include the entire range of new facilities, equipment, and test program set costs needed to support software development including compilers, operating costs, and tools. Software quality costs include all costs associated with the quality support team, including attendance at configuration audits, document reviews, and related activities. Verification and validation costs include all costs related to verification and validation by either the developer or an independent agent.

Frequency Of Reporting:

monthly

Use/Interpretation:

Cost information is indispensable to both the contractor and government program offices. MIL-HDBK-WBS.SW provides guidance in developing well defined work and cost accounting packages dealing specifically with software effort. It amplifies the requirements of MIL-STD-881B, "Work Breakdown Structure for Defense Materiel Items."

When properly identified, reported and interpreted, cost information can show how the program is progressing. Looking strictly at high level data may mask underlying problems in a lower level that will manifest themselves in large program problems downstream if they are not tended to immediately. Thus, it is prudent to require data for all potentially risky areas. However, the degree of detail must be leavened with the cost to collect and analyze the data. Finally, the Program Manager (PM) must look at the data while they are still fresh so that problems can be addressed immediately.

In financial terms, the cost data measure earned value (BCWP). Each item in the WBS is assigned a value (budgeted cost) and a time when it must be completed (schedule). As items are completed, the government receives that value for an actual cost at an actual time. If the actual cost is equal to or less than the budgeted cost, things are going well. If the actual cost is greater than the budgeted cost, it could indicate problems that require investigation.

More simplistically, BCWS is what you should have paid for what you should have gotten. BCWP is what you should have paid for what you got. ACWP is what you paid for what you got. If ACWP is greater than BCWP, the program is over cost. If BCWS is greater than BCWP the program is behind schedule. The amount of time behind schedule can be determined by determining the time distance (horizontally) between the two in Figure 7.1-1. Figure 7.1-2 displays the cost and schedule variances versus management reserve. In these examples, it should be noted that the program is over cost and behind schedule.

In many cases, costs are a good early indicator that there are problems with a particular item in the program. Excessive costs can indicate additional resources being applied to a tough issue. In software, it could indicate more programmers, higher level programmers or outside consultants being assigned to a particular CSCI than planned; it may mean more development tools or a larger programming environment than forecast; or it could mean a lot of recoding.

The data required are forward looking as well as historic. The developer is required to provide an estimate at completion for cost and schedule. The government program manager can use the same data to make his own predictions as well as to forecast the program's future well-being.

As with any data for cost and schedule, the program management office must ensure at the outset that the estimates are reasonable throughout the life of the contract. Front loading costs and back loading schedules (high costs early with most deliveries occurring late) can give early indications of success in an unhealthy program.

Properly identifying WBS categories in the contract can allow the PM to look at the resources expended overall in areas such as requirements analysis, facilities, development equipment/tools, design, coding, testing, rework, documentation, training, etc. Using a flexible data base management system could allow the PM to get a good idea of what areas do well and what areas might need improvement. For instance, the (government or contractor) PM may see a correlation between higher requirements analysis costs and lower rework costs. Or, he may see that a particular group has a considerably lower rework cost than the others; he might then want to examine their development processes to see if there are lessons that can be applied to other groups.

At the lower levels of the WBS (software activities), this metric is used to track software expenditures versus allocations over the life of a program. Status needs to be determined not only on the present percent of allocation used, but also in relation to what has been done to date and how much is yet to be done (fault profiles, breadth of testing, depth of testing, reliability, (optional) manpower, (optional) development progress). Further insight into risk can be determined by examining expenditures relating to rework (i.e., the fixing of faults and changes in requirements). Exceeding the allocation at any point in time is cause for concern and investigation.

In addition to being used with the metrics listed in the preceding paragraph, the cost metric should also be used with the detailed schedule metric, so that a comprehensive view of cost/schedule performance and status can be obtained. For example, undesirable cost trends may signal impending delays of major events or milestones.

Rules of Thumb:

For the lower level WBS software activities, management attention needs to be heightened whenever expenditures are nearing allocated values and much work is yet to be done. A program review may be necessary in this case and should be mandatory when an allocation is actually exceeded.

For higher level WBS costs, when it appears that the contractor is going to exceed his management reserve in his or the government's estimate at completion, a contractor/government level In Process Review (IPR) should be required. When it appears that either the contract is going to exceed the government's management reserve at completion, or the estimate at completion is going to breach the program baseline, a higher level government IPR should be required.

References:

"Work Breakdown Structure for Software Cost Reporting", MIL-HDBK-WBS.SW (Second Draft), 1 October 1991.

7.2 Metric: Schedule.

Purpose/Description:

The schedule metric provides indications of the changes and adherence to the planned schedules for major milestones, activities, and key software deliverables. As with the cost metric, the schedule metric takes advantage of existing acquisition policies with emphasis placed on lower level schedule considerations to ensure that key milestones, activities, and deliverables are structured and delivered in a manner that supports overall program success. It provides visibility of current and overall schedule variances, indications of likely trouble areas, and trends for the future program schedule.

Also, as with the cost metric, schedule is a primary indicator of program success or failure, and will be used in the correlation and validation process described with the cost metric.

Life Cycle Application:

Begin collecting at program start, and continue for the entire software development.

Algorithm/Graphical Display:

Plot current program schedule as shown in Figure 7.2-1. Note that this displays only the current schedule. Comparisons with previous schedules, described below, are needed to assess schedule performance.

Plot planned and actual schedules for major milestones and key software deliverables as they change over time. An example is shown in Figure 7.2-2.

In Figure 7.2-2, the Preliminary Design Review (PDR) and Critical Design Review (CDR) milestone schedules are plotted over time. Any milestone of interest can be plotted. Similar plots can also be made for key product deliverables (e.g., Software Product Specification (SPS)) as well as key activities (e.g., development of a CSCI). To read the graph, find the actual date on the x-axis, and read the appropriate planned date on the y-axis. For example, at month one, the PDR was planned for month two, and the CDR was planned for month eight. At month two, the PDR schedule has slipped to month three (a slip of one month), whereas the CDR schedule has remained the same. At month three, the PDR schedule has slipped to month five (an additional slip of two months), whereas the CDR schedule has remained the same.

Plot the planned start and end date for key activities, as they change over time. An example is shown in Figure 7.2-3.

A table showing the schedule and status of start and end dates for key present and future activities and events should be created. Table 7.2-1 serves as a representative example of such a table. A negative entry in a "slip" column indicates that the date has been moved earlier in time.

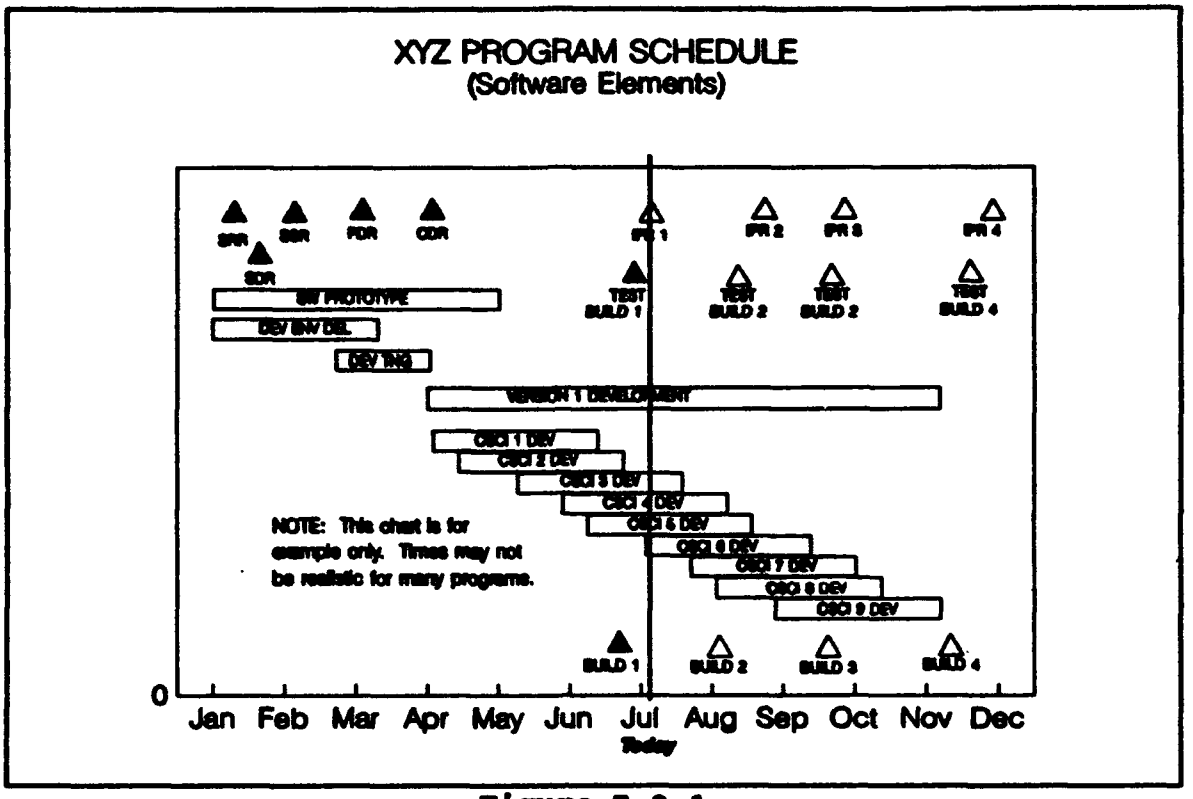

Figure 7.2-1

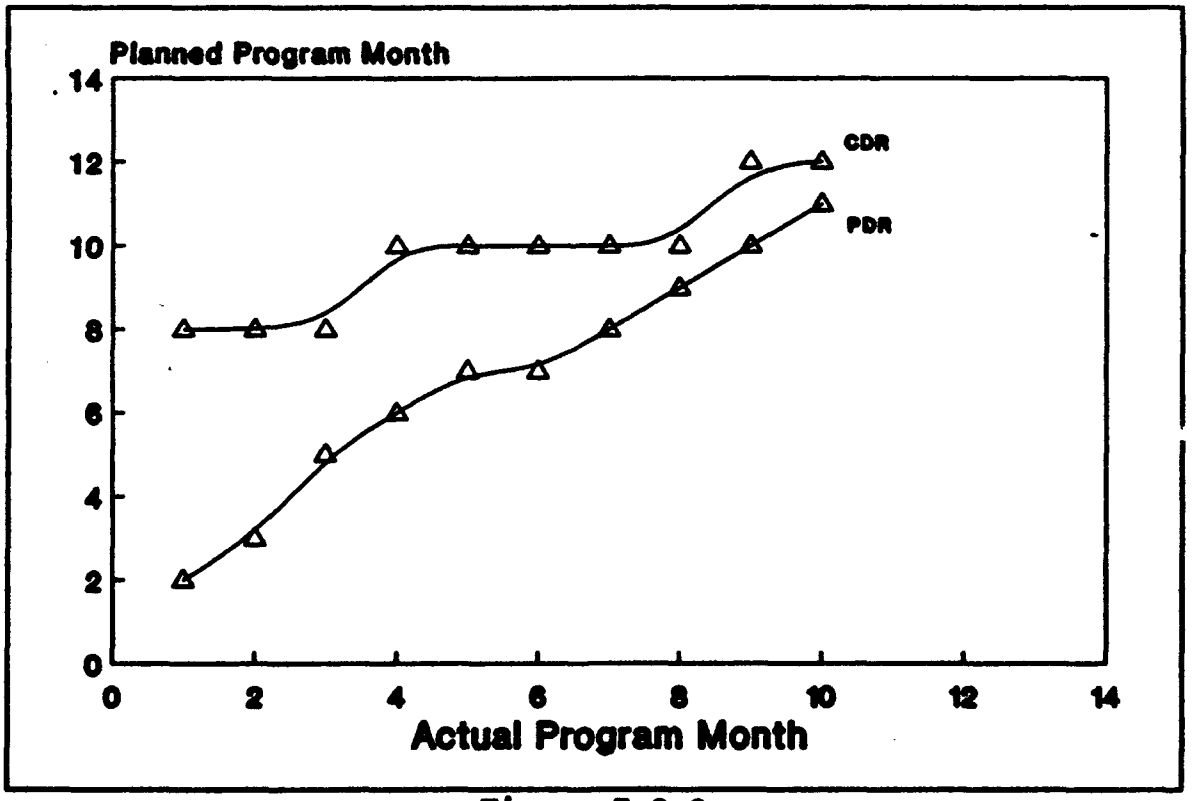

Figure 7.2-2

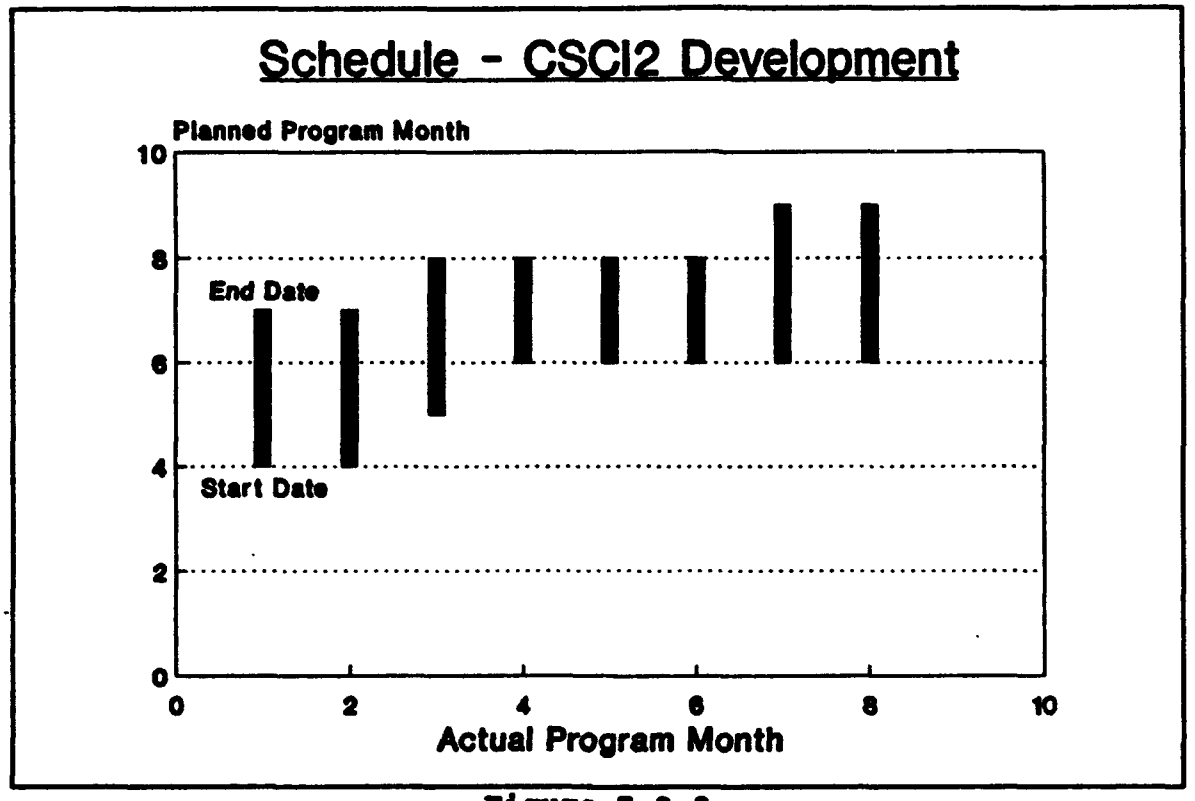

Figure 7.2-3

Table 7.2-1

Milestone, Deliverable, Activity	Latest Slip in Start Date (months)	Cumulative Slip in Start Date (months)	Latest Slip in End Date (months)	Cumulative Slip in End Date (months)
PDR	4	5	N/A	N/A
CDR	3	4	N/A	N/A
CSCI1 Development	0	0	0	0
CSCI2 Development	5	7	3	4
Integration Testing	3	4	2	2
FQT	2	3	2	3
TT	0	0	-1	-1
OT	0	0	0	0

Data Requirements:

For the purposes of these data requirements, milestones and deliverables occur at a single point in time, whereas activities span a given time period. Therefore, unlike milestones and deliverables, activities have both a start date and an end date.

for each milestone and deliverable:

(examples of milestones are System Design Review (SDR), System Requirements Review (SRR), Software Specification Review (SSR), PDR, CDR, Army System Acquisition Review Council (ASARC)/Major Automated Information System Review Council (MAISRC))
(examples of software deliverables are Software Development Plan (SDP), SPS, System/Segment Specification (SSS), Interface Requirements Specification (IRS), Version Description Document (VDD), Software Requirements Specification (SRS), Software Design Document (SDD), Software Test Plan (STP), Software Test Description (STD), Software Test Report)
initial planned date
present planned date
actual date

for each activity:

(examples of activities are CSCI development, Formal Qualification Test (FQT), Functional Configuration Audit (FCA), Physical Configuration Audit (PCA), Technical Test (TT), Operational Test (OT))
initial planned start date
present planned start date
actual start date
initial planned end date
present planned end date
actual end date

Frequency of Reporting:

monthly

Use/Interpretation:

The schedule metrics, when plotted as they change over time, provide indications of problems in meeting key events or deliveries. These metrics examine schedule considerations at a finer resolution than the cost metric.

Obviously, the higher the slope of the trend line for each milestone or event slippage (such as that shown in Figure 7.2-2), the more problems are being encountered. Milestone slippages should be investigated. Potential clustering (i.e., bunching up in time) of key events should be guarded against. Figure 7.2-2 indicates a "bunching up" of the two main design reviews (PDR and CDR). Such a condition may possibly indicate a compressed phase of activity, which may be an early warning sign of problems.

In a similar fashion, the key present and future activities should be examined using plots similar to that shown in Figure 7.2-3. Such portrayals can illuminate slips in schedule, as well as possible compressions or expansions of the activity's duration.

An additional method for analyzing the schedule for activities and events is to create and analyze a table similar to Table 7.2-1. Compression or expansion within an activity can be viewed by comparing the time allocated to it as reflected in the "latest slip" column, to the time allocated to it previously. Compression or expansion among events and activities can be seen by looking at relative start and end dates between them.

The schedule metric should be used in conjunction with several other metrics to help judge program risk. For example, it should be used with the test coverage metrics to determine if there is enough time remaining on the current schedule to allow for the completion of all testing.

The schedule metric passes no judgement on the achievability of the developer's initial schedule.

Rules of Thumb:

No formal rules for the trend of the schedule metrics are given. However, large slippages or compressions are indicative of problems. Maintaining conformance with calendar driven schedules should not be used as the basis for proceeding beyond milestones.

References:

None.

7.3 Metric: Computer Resource Utilization (CRU).

Purpose/Description:

This metric is intended to show the degree to which estimates and measurements of the target computer resources (central processing unit (CPU) capacity, memory/storage capacity, and input/output (I/O) capacity) used are changing or approaching the limits of resource availability and specified constraints. Overutilization of computer resources can have serious impacts on cost, schedule, and supportability. Approaching resource capacity may necessitate hardware change or software redesign. Exceeding specified reserve requirements can have similar impacts in the post deployment phase. Proper use of this metric can also assure that each resource in the system has adequate reserve to allow for future growth due to changing or additional requirements without requiring redesign. This metric can be applied to a system architecture which is distributed or centralized.

Life Cycle Application:

Early in the design phase, utilization budgets should be established for each processor and I/O channel in the system. Memory/storage usage budgets should be allocated to all computer software units (CSUs) (i.e., the lowest design element that is separately testable), computer software components (CSCs), computer software configuration items (CSCIs), and temporary and permanent data files early in the design phase. Based on these estimates, each device (CPU, I/O, and memory/storage) should be sized so that only half of the available capacity of the device is utilized. These targets should be documented in the SSS and analyzed with respect to current estimates at each design review.

All changes to these initial estimates should be reported, including those caused by hardware modifications. For the memory and I/O categories, actual usage should be measured monthly during coding, unit testing, integration testing, CSCI testing and system level testing. For the CPU usage statistic, measurements should be taken monthly after the beginning of unit testing. Actual utilization should be formally demonstrated at the system level for each resource under peak loading conditions during FQT, and during Post Deployment Software Support (PDSS) if additional capability is added.

Algorithm/Graphical Display:

The allocation for each resource type should not exceed the target upper bound utilization for any category (for some systems, the allocation equals the target upper bound).

CPU and I/O resource utilization are typically measured by the system. While the measurements contribute slightly to system overhead, the feature typically comes with the system in its off the shelf configuration. In instances where the system does not measure itself in term of CPU and I/O utilization, the percent utilization must be measured using appropriate instrumentation or test hooks; it is recognized that these tools may add to system overhead.

For memory/storage resources, the percent utilization must be computed. For memory, the resource is random access memory (RAM), and usage must be measured with dynamic analysis tools. For storage, the resources include disk space and other mass storage. For those elements of the software that do not change (e.g., the source code in terms of CSUs, CSCs), measurement can be easily done with straightforward operating system commands that measure the amount of used and free space on a device. For those elements of the software that change (e.g., temporary and permanent data files), these resources must be measured on the fly with dynamic analysis tools or with the periodic, interrupt-driven use of static tools. Again, the percent utilization should be computed with respect to the devices, and not the components that happen to be stored on those devices.

Local Area Networks (LANs) and data buses should also be considered as resources; data should be recorded and reported separately as with any other resource.

As software development proceeds, the measured values for each category should be projected out to the "full" system. For example, if half of the "size" of the software is built and measured, the projected value for utilization would be the actual for the portion built and measured to date plus the budgeted portion yet to be added.

In Figure 7.3-1, target upper bound utilization is shown as a straight line. In reality, the target upper bound utilization can change over time. The sample shown represents the utilization of a single CPU resource; similar graphs or tables should be constructed for the utilization of all other CPUs plus each I/O and memory resource.

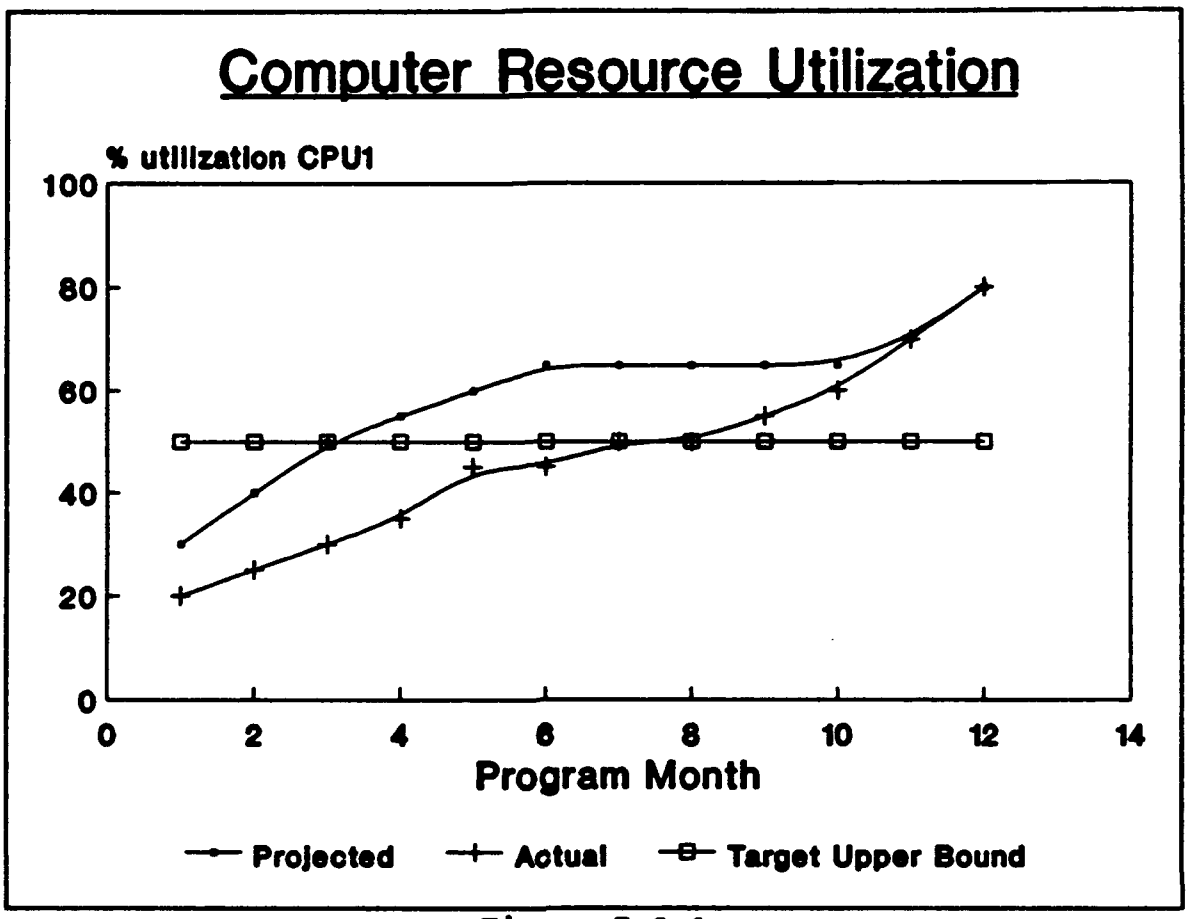

Figure 7.3-1

Data Requirements:

(Notes:

1. Usage should be measured during peak operational loading periods and should include the operating system and non-developer supplied software as well as the development software.
2. Where "target" is used in these data elements, it is actually meant to construe the target upper bound.)

date of report
for each CPU:
- unique identifier
- initial target CPU usage (percent of capacity)
- present target CPU usage (percent of capacity)
- actual CPU usage (percent of capacity)
- projected CPU usage (percent of capacity)

for each I/O channel (and LAN or data bus as appropriate):
- unique identifier
- initial target I/O usage (percent of capacity)
- present target I/O usage (percent of capacity)

actual I/O usage (percent of capacity)
projected I/O usage (percent of capacity)

for each RAM memory resource:
unique identifier
capacity of resource (in bytes)
initial target upper bound (in bytes)
present target upper bound (in bytes)
actual usage (in bytes)
projected usage (in bytes)

for each mass storage device:
unique identifier
capacity of device (in bytes)
initial target upper bound (in bytes)
present target upper bound (in bytes)
actual usage (in bytes)
projected usage (in bytes)

Frequency of Reporting:

initial targets : monthly starting with SSR
all other values : monthly starting with CDR

Use/Interpretation:

Resource utilization tends to increase over the development of a project. Therefore, adequate planning must be done up front to ensure that the software's operation does not put undue demands on the target hardware's capabilities. This measure allows one to track utilization over time to make sure that target upper bound utilization is not exceeded and that sufficient excess capacity remains for future growth and for periods of high stress loading.

In multiprocessor environments, each processor should be targeted and tracked separately.

Tailoring may be appropriate for situations when dynamic allocation, virtual memory, parallel processing, multitasking or multi user-based features are employed.

In instances where the development and target environments differ in types and/or capacities, caution must be taken in computing and analyzing the measures. Translations are acceptable up to a certain point, but testing on the target hardware must take place as early as possible.

Initial estimates should be retained for comparison with what is finally achieved in order to aid in scoping future programs.

In addition to collecting utilization data throughout the build up of contractor testing (including single thread to multiple thread), measurements should also be taken during system level stress testing.

During development, it is important to look at both actual and projected values in relation to the target upper bound values. If either exceeds the target values, extra attention should be paid to assure that the projections decay to below the target upper bound value by project completion. If it is apparent from the projections that the target upper bound limits will be exceeded, action must be taken to either optimize the software or upgrade the capability of the target configuration.

It should be noted that sudden drops in utilization may reflect either new system capacity or new software that embodies more efficient programming.

Computer resource utilization metrics should be used in conjunction with the test coverage/success metrics (breadth and depth of testing) to ensure that measures of the actual usage are representative and portray the entire system under realistic stress loads. If the optional development progress metric is available, computer resource utilization metrics should be used in conjunction with development progress to ensure remaining development can be accommodated without exceeding planned utilization.

Computer resource utilization provides the link to total system performance. As mentioned previously, on many computer platforms, the data are relatively easy to collect (indeed, often self-measured), and are often built in to the overhead of the system. On other platforms, however, the data may have to be collected, which will contribute some small amount to system overhead.

Rules of Thumb:

For embedded/tactical systems, design for no more than 50 percent utilization for memory/storage, CPU, and I/O resources. For information area systems, a higher target upper bound value may be allowable, but should be specified in the requirements documents. Performance may deteriorate when utilization exceeds 70 percent for time critical applications. Schedule and cost can be severely impacted as utilization exceeds 90 percent.

For systems employing virtual memory architectures, usage of RAM is not as important as measuring the amount of swapping that occurs during peak load periods.

If, at any time during development, actual or projected computer resource utilization exceeds target upper bound utilization, an immediate review must be held. Corrective action (e.g., software redesign, hardware upgrades, etc.) must be taken before proceeding to the next stage of development.

References:

"Software Reporting Metrics", The Mitre Corporation, ESD-TR-85-145, November 1985.

"Software Management Indicators", Air Force Systems Command, AFSCP 800-43, January 31, 1986.

7.4 Metric: Software Engineering Environment.

(Note: Much of the information below has been extracted directly from the referenced SEI report).

Purpose/Description:

The software engineering environment rating is used to provide an indication of the developer's typical use of modern, accepted software engineering principles (e.g., the use of structured design techniques, the extent of tool usage, the use of program design languages (PDL), etc.) in the development of software. If practical, aspects of the methodology could also be applied to materiel developer personnel or the program manager's matrix support staff for the purpose of assessing capabilities with respect to software development.

SEI has defined four types of evaluations/assessments; self, government, SEI-assisted, and commercial. The most rigorous, and therefore the most desired, are either the government or SEI-assisted evaluations. However, they are also the most costly, and may not be practical for small programs. Therefore, any of the four are acceptable, but should be performed for each subdeveloper working on the project.

Life Cycle Application:

At each major milestone where developers will be selected or assigned.

Algorithm/Graphical Display:

Follow SEI methodology, which includes the following:
Collect questionnaire data from developer.
Conduct follow up visit (assessment team) to answer further questions, observe tools, etc.
Perform assessment.
Calculate process maturity levels, which are broadly defined as possessing the following characteristics :

1. Initial -
   ill-defined procedures and controls
   no consistent application of software engineering management to the process
   no use of modern tools and technology

2. Repeatable -
   management of costs and schedules
   use of standard methods and practices for managing some software development activities

3. Defined -
   software development process well-defined in terms of software engineering standards and methods
   increased organizational focus on software engineering

use of design and code reviews
internal training programs
establishment of software engineering process group

4. Managed -
software development process is quantified, measured, and well-controlled
decisions are based on quantitative process data
use of tools to manage design process
use of tools to support data collection and analysis
accurate projection of expected errors

5. Optimizing-
major focus on improving and optimizing process
sophisticated analysis of error and cost data
employment of error cause analysis and prevention studies
iterative improvement of process

Data Requirements:

Name of (sub)developer.
Type of assessment/evaluation.
Results of SEI rating (numerical rating and list of key process areas).

Frequency of Reporting:

Once for each developer selection process. If desired, within a long development phase, additional ratings can be performed if desired (e.g., if a phase of Engineering & Manufacturing Development has been ongoing for 5 years, and will be continuing for several more years, perhaps another evaluation or assessment of the developer(s) should be performed).

Use/Interpretation:

The software engineering environment rating provides a consistent measure of the capability of a developer to use modern software engineering techniques in his development process, and therefore his capability to instill such principles and characteristics in the product. Obviously, it can be seen from the definition of each rating level that higher is better. The basic assumption to this approach is that a quality process results in a quality product. The other eleven metrics, as well as other evaluation techniques, should be used to examine the quality of the product.

The primary use of the software engineering environment rating is during the source selection process. However, besides the use as a tool with which to relatively compare the ability of developers, the use of the software engineering environment rating may encourage contractors to improve their software development process in order to increase their rating. A higher rating will increase the developer's chance of being selected for future software development projects.

In addition to the numerical rating which summarizes the developer's software process maturity, the subelements which comprise the rating, called key process areas, can be examined to determine relative strengths and weaknesses within a rating band. Additionally, the developer may have demonstrated capability on certain individual process areas that are indicative of a higher rating level. The weaker process areas are those which should be targeted for improvement in order to move forward to a higher rating.

It is commonly believed that a metrics program will be cheaper to implement for developers who possess a high SEE rating.

Rules of Thumb:

On a relative basis, the process maturity levels of various contractors can be compared. The SEI reports that currently, only a very small percentage (i.e., 2 or 3 %) of companies have achieved ratings of level 3, 4, or 5. Most companies are rated at level 1 or 2.

References:

"A Method for Assessing the Software Engineering Capability of Contractors", Carnegie-Mellon University Software Engineering Institute Technical Report CMU/SEI-87-TR-23, September 1987.

7.5 Metric: Requirements Traceability.

Purpose/Description:

The requirements traceability metrics are used to measure the adherence of the software products (including design and code) to their requirements at various levels. It also aids the combat developer, materiel developer, and evaluators in determining the operational impact of software problems.

Life Cycle Application:

Begin tracing during user requirements definition phase, in support of requirements reviews, PDR, CDR, milestones, and major releases.

Algorithm/Graphical Display:

This metric is a series of percentages, which can be calculated from the matrix described below.

Trace all Operational Requirements Document (ORD) requirements for automated capabilities to the Users' Functional Description (UFD). Identify any ORD requirements for automated capabilities not found in the UFD. Calculate the percentage of ORD requirements for automated capabilities in the UFD.

Trace all UFD requirements by priority to the system specification (SS) (either a SSS, Prime Item Development Specification, or Critical Item Development Specification). Identify omissions. Calculate the percentage of UFD requirements that are in the SS.

Trace all software-related SS requirements to the SRS(s) and IRS(s). Identify omissions.

The software requirements as specified in the SRS(s) and IRS(s) must then be traced into the software design, code, and test cases. The other percentages that must be calculated are:

- % software requirements in the CSCI design
- % software requirements in the CSC design
- % software requirements in the CSU design
- % software requirements in the code
- % software requirements which have test cases identified

The technique to be used in performing this analysis is the development of a software requirements traceability matrix (SRTM). The SRTM is the product of a structured, top-down hierarchical analysis that traces the software requirements through the design to the code and test documentation. The SRTM should contain information similar to Table 7.5-1. A dendritic numbering is shown for some columns. The software specifications and requirements should be listed in groups which represent higher order system requirements. In this manner, the grouping of CSUs which represent a required system function can be readily seen. Also, it is good practice to trace the requirements at additional levels between design and code. For example, they can be traced to functional decomposition documents, flow diagrams, data dictionaries, etc.

Table 7.5-1

SOFTWARE REQUIREMENTS TRACE MATRIX

ORD Reqts	UFD		SS Reqts	SRS Reqts	IRS Reqts	Design			Code	Test Case
	Reqts	Prior.				CSCI	CSC	CSU		
r1	1.1.1	1	1.1.2	SRS_1.r1	IRS_1.r1	$CSCI_1$	CSC_1	CSU_1	CSU_1	1.1.1
						$CSCI_1$	CSC_1	CSU_4	CSU_4	?
r1	1.1.2	2	2.3.3	SRS_1.r2		$CSCI_1$	CSC_2	CSU_3	CSU_3	1.1.2
r1	1.2.1	1	3.8.1	SRS_2.r3		$CSCI_2$	?	?	?	?
r2	.	.	.	.		.	.	.	.	.
.	.	.	.	.		.	.	.	.	.
.	.	.	.	.		.	.	.	.	.

Notes:
1) A question mark indicates that tracing has not yet occurred.
2) A blank entry indicates that tracing is not applicable, and should not be counted as part of any percentage.
3) At each level of the trace, a single requirement can be traced to multiple lower level requirements.

The SRTM will be completed to various degrees depending upon the current stage of the software life cycle and should be part of the Technical Data Package. From the SRTM, statistics can be calculated indicating percentage of tracing to various levels.

In some instances, it might be interesting to perform a backwards trace (e.g., from code to requirements). In lieu of creating another matrix, one can simply make a list of the distinct entries and compare with a total count of the entries for the column of interest. To carry the example of doing a backwards trace from code to requirements further, one would make a list of all the distinct CSUs which appear in the "code" column of the SRTM. This list should then be compared with the total list of CSUs for the system. Any CSU which does not appear in the "code" column may not support any requirement. These CSUs should be investigated.

Data Requirements:

list of requirements and design specifications
(ORD, UFD, SS, SRS, IRS, SDD)
completed SRTM
software test description in accordance with DOD-STD-2167A
number of ORD requirements for automated capabilities:
total
traceable to UFD
not traceable to UFD
number of UFD requirements:
total
traceable to SS
not traceable to SS
number of SS software requirements:
total
traceable to SRS/IRS
not traceable to SRS/IRS
for each CSCI, number of SRS requirements:
total
traceable to CSCI design
traceable to CSC design
traceable to CSU design
traceable to code
having test cases for all of its CSUs
number of SRS requirements not traceable to UFD

Frequency of Reporting:

update periodically in support of requirements reviews, milestones and major releases.

Use/Interpretation:

As can be seen in the algorithm above, the tracing of requirements must occur at several levels. This tracing should be a key government tool at all system requirement and design reviews. It can serve to indicate those areas

of requirements or software design that have not been sufficiently thought out. The trend of the SRTM should be monitored over time for closure.

By the nature of the software development process, especially in conjunction with an evolutionary development strategy, the tracing of requirements will be an iterative process. That is, as new software releases add more functionality to the system, the trace of requirements will have to be revisited and augmented.

Although not portrayed graphically above, trends of requirements traceability can be shown over time. For example, during the requirements phase, the percent of UFD requirements traced into the SRS can be depicted over time.

One of the important new characteristics that will be embodied in the UFD is a grouping of requirements by four priority levels. Such a prioritization should be used with the SRTM to highlight certain key user functions. In cases where there are vast numbers of requirements, a common sense approach would be to attack the UFD priority one requirements first.

Another benefit of requirements traceability is that those modules which appear most often in the matrix (thus representing the ones that are most crucial in the respect that they are required for multiple functions or requirements) can be highlighted for earlier development and increased test scrutiny.

The requirements traceability metrics should be used in conjunction with the test coverage metrics (depth and breadth of testing). It is important to note that tracing to a test case passes no judgement on the sufficiency of the test case(s). Rather, an entry in the SRTM only indicates that at least one test case exists. Further, the SRTM does not address the conduct or outcome of any testing. Breadth and depth of testing must be used to glean test coverage and success for particular requirements. The requirements traceability metrics should also be used with the (optional) development progress metric to verify if sufficient functionality has been demonstrated to warrant proceeding to the next stage of development or testing. They should also be used in conjunction with the design stability and requirements stability metrics.

Due to the detailed nature of these requirements traceability metrics, they should be a normal product of the V&V effort. The SRTM may be generated by the developer, but it must be verified by an independent agent such as the IV&V contractor, with coordination from the Test Integration Working Group (TIWG) and Computer Resources Working Group (CRWG).

During PDSS, if a function is modified, the SRTM can be used to focus regression testing on a particular CSCI/CSC/CSU.

Finally, it should be noted that some software requirements that are qualitative in nature (e.g., user friendliness) cannot be traced to specific design and code. It remains paramount, however, that these requirements be evaluated by other means.

Rules of Thumb:

Do not approve UFD until all ORD requirements for automated capabilities have been traced into it.

Do not proceed beyond SSR until all UFD requirements have been traced into the SS.

Do not proceed beyond CDR until a high trace percentage exists from the ORD to the design at CSCI, CSC, and CSU level.

Do not proceed to formal government testing until, at a minimum, all priority one requirements have been traced into the code and have test cases identified.

References:

Operational Requirements for Automated Capabilities, Draft DA Pamphlet XX-XX, 2 January 1992.

7.6 Metric: Requirements Stability.

Purpose/Description:

The metrics on requirements stability indicate the degree to which changes in the software requirements or changes in the developer's understanding of the requirements affect the software development effort. It also allows for determining the cause of requirements changes.

Life Cycle Application:

Begin collecting during user requirements definition phase. Measure requirements with respect to the UFD before Milestone (MS) II. Measure requirements with respect to the SRS/IRS after MS II.

Algorithm/Graphical Display:

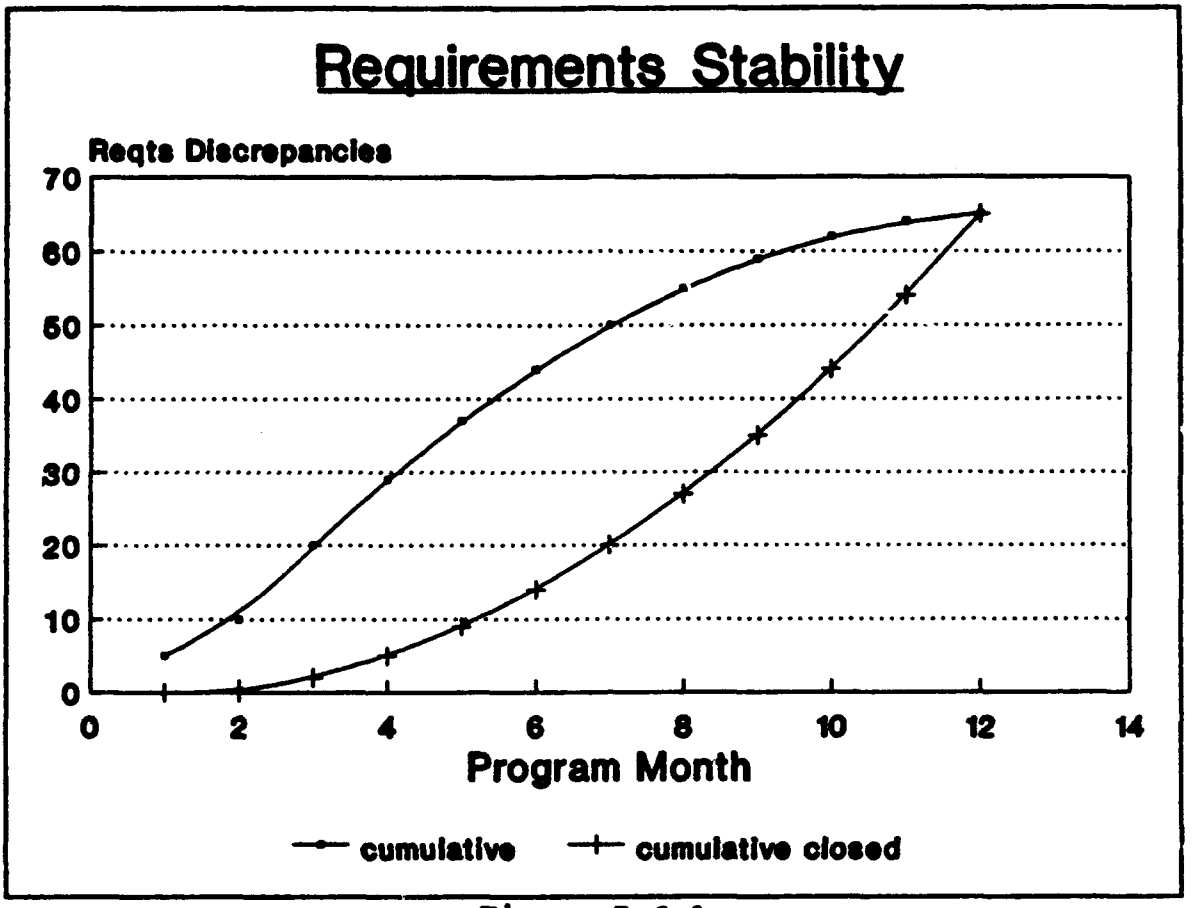

Figure 7.6-1

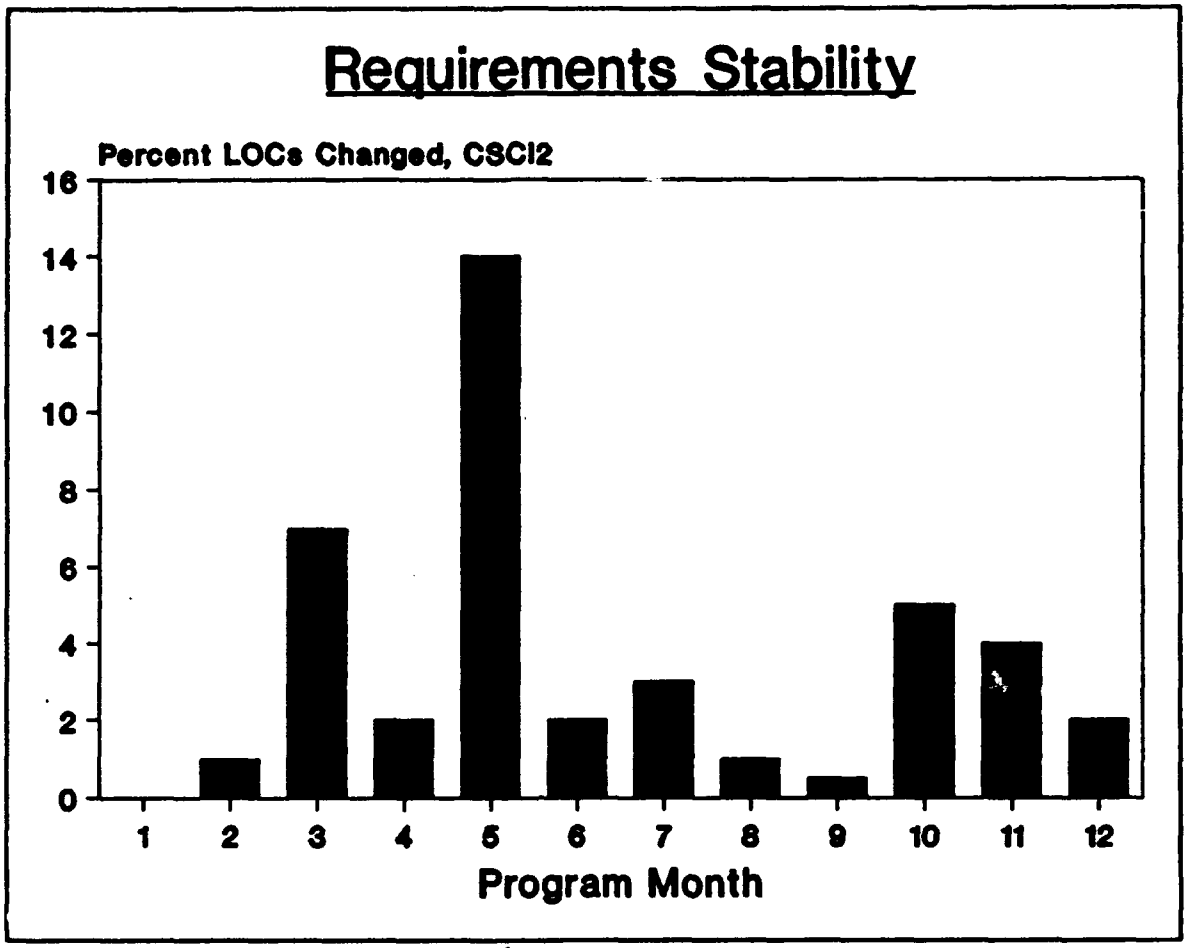

Figure 7.6-2

Figure 7.6-1 shows cumulative requirements discrepancies (number of requirements added + number of requirements deleted + number of requirements changed) over time versus closure on those discrepancies. Figure 7.6-2 is a representation of the effect of requirements discrepancies on the code (percent of lines of code (LOC) changed by month). In actuality, one wants to develop several versions of the second chart. One version should show the number of user-initiated requirements discrepancies and the number of developer-initiated requirements discrepancies. The second version should show the percent LOC affected by user-initiated discrepancies and the percent LOC affected by developer-initiated discrepancies. Additionally, one might want to look at the number of modules affected by both types of requirements discrepancies.

Data Requirements:

number of requirements discrepancies
- number of software requirements added by user
- number of software requirements deleted by user
- number of software requirements modified by user

number of software requirements added by developer
number of software requirements deleted by developer
number of software requirements modified by developer
monthly status (cumulative total and total number resolved) of above discrepancies
number of modules affected by user-initiated requirements discrepancies
number of modules affected by developer-initiated requirements discrepancies
total number of LOCs
for each CSCI
number of LOC affected by approved Engineering Change Proposals - Software (ECPs-S) due to user-initiated requirements discrepancies
number of LOC affected by approved ECPs-S due to developer-initiated requirements discrepancies

Note: For definition of LOC see Complexity metric.

Frequency of Reporting:

monthly and at major reviews

Use/Interpretation:

When a program is begun, the details of its operation and design are rarely complete, so it is normal to experience changes in the specifications as the requirements become better defined over time. (Note: it is believed that rapid prototyping can help to alleviate this problem, or at least to cause the refinement to happen early in development). When design reviews reveal problems or inconsistencies, a discrepancy report is generated. Closure is accomplished by modifying the design or the requirements. When a change is required that increases the scope of the project, an ECP-S is submitted.

The plot of open discrepancies can be expected to spike upward at each review and to diminish thereafter as the discrepancies are closed out. For each engineering change, the amount of software affected should be reported in order to track the degree to which ECPs-S increase the difficulty of the development effort. Only those ECPs-S approved by the configuration control board should be counted. Good requirements stability is indicated by a leveling off of the cumulative discrepancies curve with most discrepancies having reached closure.

The later in development a requirements discrepancy occurs, the more impact it has on the program. Those discrepancies which occur before design and coding don't affect nearly as much as those which occur after design has started. Those discrepancies which occur during design can impact previous design work. The worst case is when both design and code are affected. The impact of the changes can be seen by looking at the number of LOCs changed. The cause of these changes can be evaluated by examining both Requirements Stability and Design Stability together. If design stability is low and

requirements stability is high, the designer/coder interface is suspect. If design stability is high and requirements stability is low, the interface between the user and the design activity is suspect. If both design stability and requirements stability are low, both the interfaces between the design activity and the code activity and between the user and the design activity are suspect.

The metrics for requirements stability should also be used in conjunction with those for requirements traceability, fault profiles, and the (optional) development progress.

Allowances should be made for higher instability in the case where rapid prototyping is utilized. At some point in the development effort, the requirements should be firm so that only design and implementation issues will cause further changes to the specification.

As mentioned previously, it is recognized that LOC is somewhat dependent on both the application language as well as the style of the programmer. The key is to watch for significant changes to the measure over time, given that consistent definitions are used.

Rules of Thumb:

No formal values are given, but a high level of instability at the CDR stage indicates serious problems that must be addressed prior to proceeding to coding. For MSCR systems, requirements stability should be high by MS II.

References:

"Software Reporting Metrics", The Mitre Corporation, November 1985.

7.7 Metric: Design Stability.

Purpose/Description:

Design stability is used to indicate the amount of change being made to the design of the software. The design progress ratio shows how the completeness of the design is progressing over time and helps give an indication of how to view the stability in relation to the total projected design.

Life Cycle Application:

Begin tracking at PDR and continue for each version until completion.

Algorithm/Graphical Display:

A design related change is defined as a modification to the software that involves a change in one or more of the following:

1) algorithm
2) "units" (i.e., kilometers, miles)
3) interpretation (definition) of a parameter
4) the domain of a variable (i.e., range of acceptable values)
5) interaction among CSUs
6) more than one CSU

M = number of modules in current delivery/design
F_c = number of modules in current delivery/design that include design related changes from previous delivery
F_a = number of modules in current delivery/design that are additions to previous delivery
F_d = # modules in previous delivery/design that have been deleted
T = total modules projected for project

$$S \text{ (stability)} = [M - (F_a + F_c + F_d)] / M$$

$$DP \text{ (design progress ratio)} = M/T$$

Notes:

1. Although not indicated in Figure 7.7-1, it is possible for stability (S) to be a negative value. This may indicate that everything previously delivered has been changed and more modules have been added or deleted.
2. If some modules in the current delivery are to be deleted from the final delivery, it is possible for design progress (DP) to be greater than one.

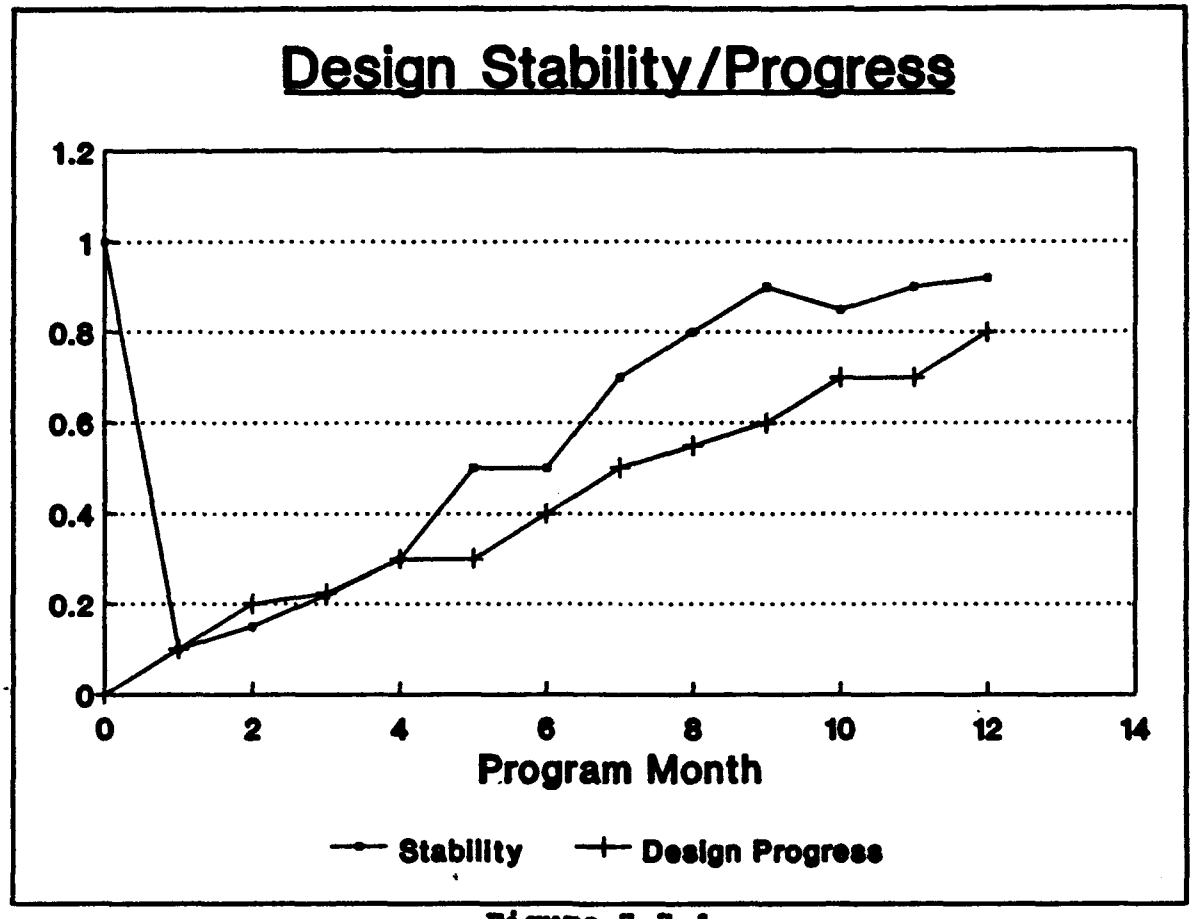

Figure 7.7-1

Data Requirements:

For each version number:
- date of completion
- M
- F_c
- F_a
- F_d
- T

Frequency of Reporting:

monthly or at each delivery

Use/Interpretation:

Design stability should be monitored to determine the number and potential impact of design changes, additions, and deletions on the software configuration. The trend of design stability over time and releases provides an indication of whether the software design is approaching a stable state, that is, a leveling off of the curve at a value close to or equal to one. In

addition to a high value and level curve the following other characteristics of the software should be exhibited:

- requirements stability is high
- depth of testing is high
- the fault profile curve has leveled off and most Software Trouble Reports (STRs) have been closed
- the (optional) development progress metric is high

Caution must be exercised, however, due to the fact that this metric does not measure the extent or magnitude of the change within a module.

The higher the stability, the better the chances of a stable software configuration. However, a value close to one is not necessarily good unless M is close to the total number of modules required in the system (DP approaching 1), and the magnitude of the changes being counted are relatively small and diminishing over time; without so doing, periods of inactivity could be mistaken for stability.

When changes are being made to the software, the impact on previously completed testing must be assessed.

Allowances for exceptional behavior of this metric should be made for the use of rapid prototyping. It is thought that rapid prototyping, while possibly causing lower stability numbers (i.e., higher instability) early in the program, will positively affect the stability metric during later stages of development.

It should be noted that not all changes made to the software are design related. For example, the insertion of additional comments within the code does not change the design of the software.

The design stability metric can be used in conjunction with the complexity metric to highlight changes to the most complex modules. It can also be used with the requirements metrics to highlight changes to modules which support the most critical user requirements.

Finally, it must be pointed out that the design stability metric does not assess the quality of the design. Other metrics (e.g., complexity) can contribute to such an evaluation.

Rules of Thumb:

No hard and fast values are known at this time. However, allowances should be made for lower stability in the case of rapid prototyping or other development techniques that do not follow the standard waterfall model for software development. In either case, an upward trend with a high value for both stability and design progress is recommended before acceptance for government testing. A downward trend should be cause for concern.

Experiences with similar projects should be used as a basis for comparison. Over time, potential thresholds may be developed for similar types of projects.

References:

"Draft Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software", IEEE Computer Society, May 1988.

7.8 Metric: Complexity.

Purpose/Description:

Complexity measures give an indication of the structure of the software and provide a means to measure, quantify and/or evaluate the structure of software modules. It also indicates the degree of unit testing which needs to be performed. It is commonly believed that the more complex a piece of software is, the harder it is to test and maintain. Additionally, it is widely felt that a highly complex module is more likely to contain embedded errors than a module of lower complexity. Accordingly, lower complexity ratings reflect software that is easier to test and maintain, thus logically resulting in fewer errors and lower life cycle costs.

McCabe's cyclomatic complexity metric measures the internal structure of a piece of software.

Halstead's metrics estimate a program's length and volume based on its vocabulary (operators and operands).

Other simpler complexity metrics are control flow, the number of executable lines of code per module (relates to the ease of understanding and maintaining the module), and the percent comment lines.

Life Cycle Application:

Begin collecting McCabe's cyclomatic complexity metric at PDR stage. Begin collecting other complexity metrics at CDR, as the modules are placed under developer configuration control. Revisit during PDSS activities.

Algorithm/Graphical Display:

(Note: As used below, the term "module" is meant to be equivalent to CSU.)

McCabe cyclomatic complexity metric :

Let E = # of edges (program flows between nodes; i.e., branches)
N = # of nodes (groups of sequential program statements)
P = # of connected components (on a flow graph, it is the number of disconnected parts)

Compute:

Cyclomatic Complexity : $C = E - N + 2P$

The quickest way to gain a basic understanding of the cyclomatic complexity metric is to graphically portray the structure of a module. Figure 7.8-1 depicts a flow graph of a module, along with its associated complexity calculations.

There are additional ways of calculating cyclomatic complexity. One such way is to calculate the number of control tokens + 1. Control tokens are programming language statements which in some way provide decision points which modify the top-down flow of the program. In other words, statements

such as IF, GOTO, CASE, etc., are considered to be control tokens since they base program flow upon a logical decision thereby creating alternative paths which program execution may follow. A CASE statement would contribute (N - 1) to complexity, where N is the number of conditions or cases associated with the statement.

Figure 7.8-2 portrays a histogram of a CSCI's modules by complexity.

Halstead metric :

Let n_1 = # distinct operators
n_2 = # distinct operands
N_1 = total # occurrences of the operators
N_2 = total # occurrences of the operands

Compute:

Vocabulary : $v = n_1 + n_2$
Program Length : $L = N_1 + N_2$
Volume : $V = L (\log_2 v)$

Control flow metric :

Count the number of times in each module where control paths cross ("knots"). (For example, a GOTO statement would cause a knot to occur in the module's flow graph).

Non-comment, non-blank, executable and data statements (herein referred to as LOC) per module metric :

count the number of non-comment, non-blank executable and data statements in each module.

Percent comment lines metric :

Let C = # comment lines in module
T = total # non-blank lines in module

Compute :

Percent comment lines = (C / T) * 100

Table 7.8-1 summarizes the various subelements of the complexity metric:

Table 7.8-1

Metric	Measure
Cyclomatic Complexity	# independent paths
Halstead	volume (operators/operands)
Control Flow	# times paths cross
LOC	size
% comment lines	degree of self-documentation

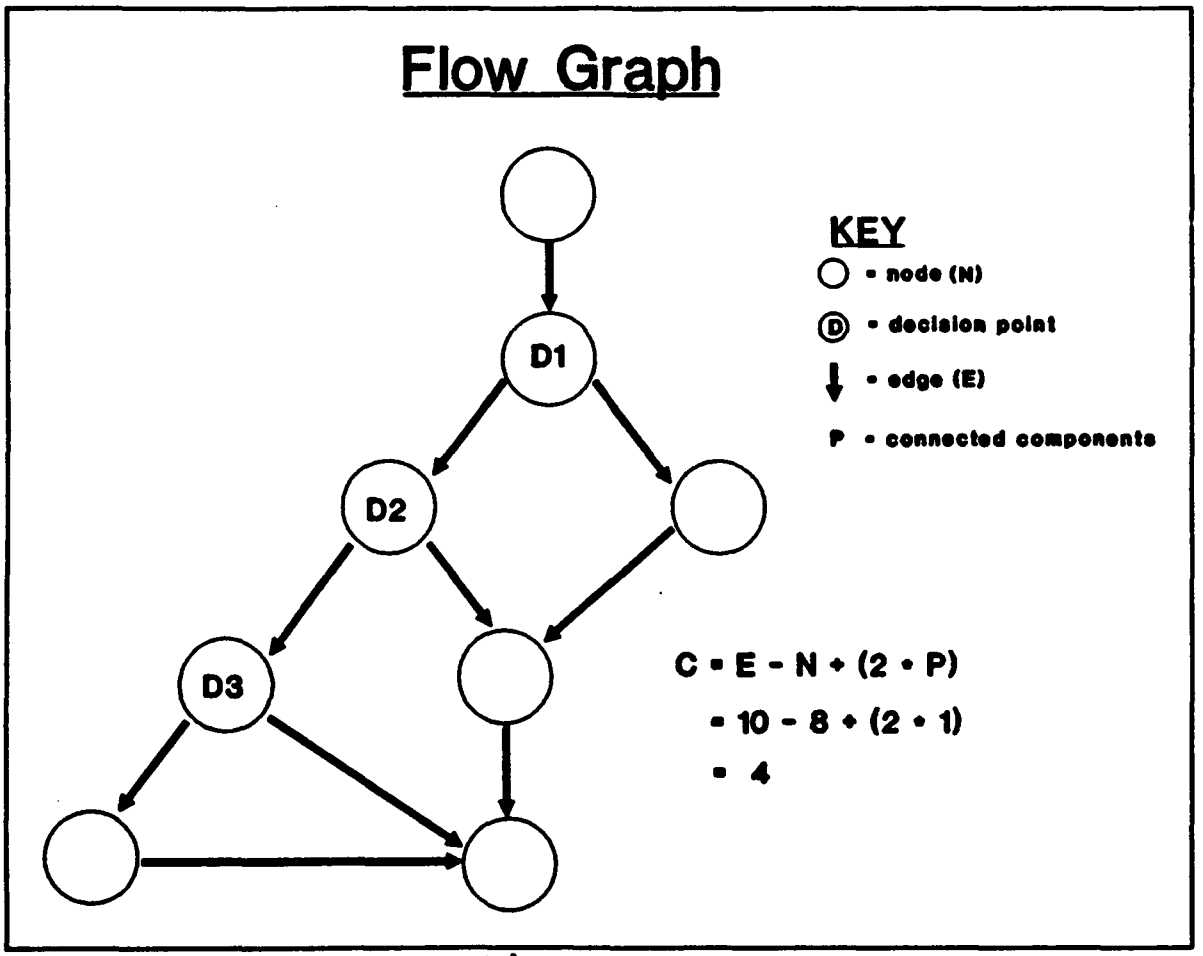

Figure 7.8-1

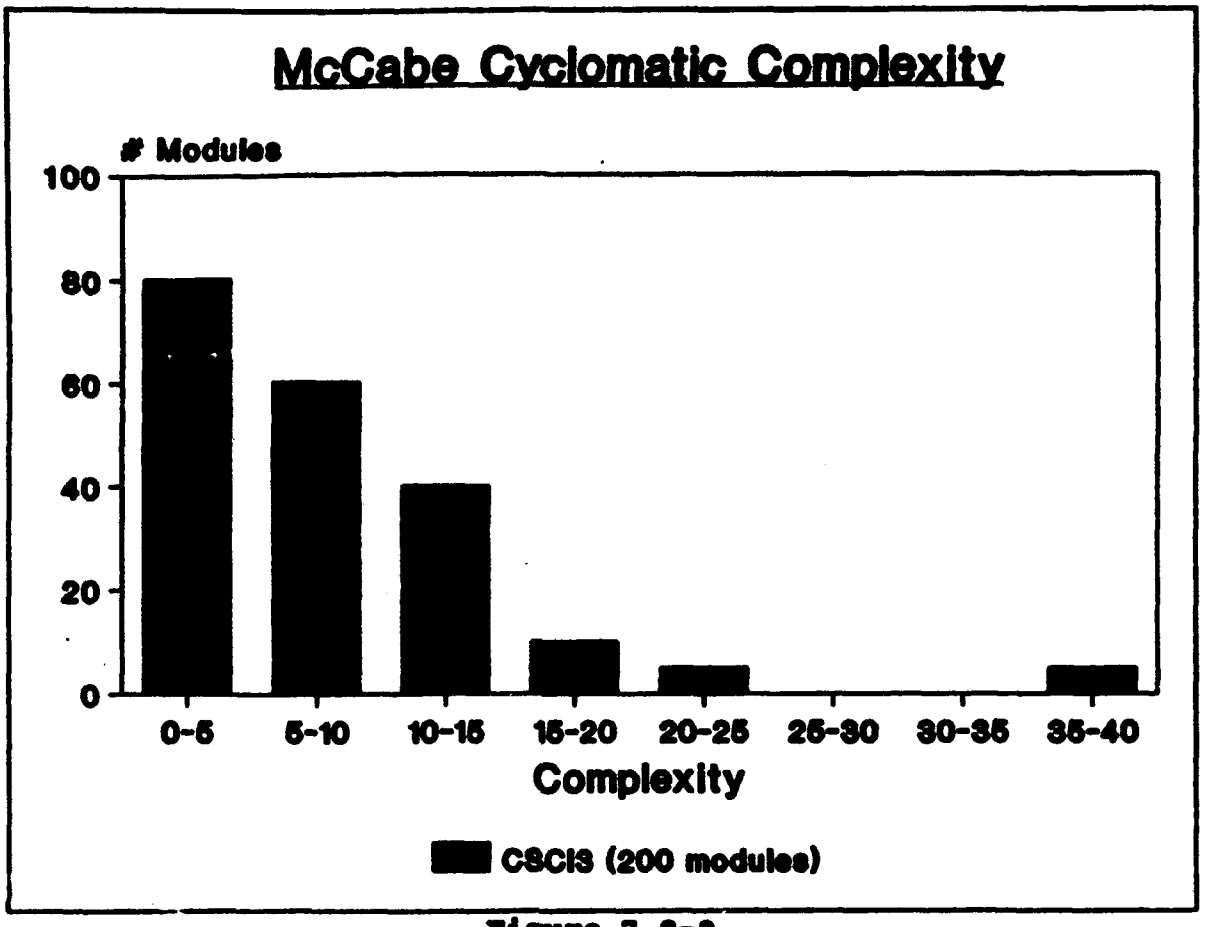

Figure 7.8-2

Data Requirements:

for each module
language
version number of language
source code
number of nodes
number of edges
number of connected components
number of distinct operators
number of distinct operands
total number of occurrences of the operators
total number of occurrences of the operands
number of times in each module where control paths cross
number of LOCs per module
percent comment lines
all complexity input data and results of calculations listed above
each of the above complexity values

Frequency of Reporting:

monthly, for any module that has changed.

Use/Interpretation:

Automated tools are available and should be used to assist in the computation of the complexity measures. There are tools to construct the flow graph depicted in Figure 7.8-1; other tools are available to compute the various complexity metrics.

This metric is used throughout the software life cycle. Requiring the complexity limit as a contractual requirement will stimulate structured programming techniques, thereby impacting design by limiting the number of "basis" paths in a program at the design and coding stages. It is used during software testing to identify basis paths (i.e., critical paths), to define and prioritize the testing effort, and to assess the completeness of CSU testing. During the maintenance phase, a proposed change should not be allowed to substantially increase the complexity, thereby increasing the testing effort and decreasing maintainability.

The complexity metrics should be generated for each module in the system. The metrics can be partitioned in several ways (e.g., by CSCI). In these partitionings, one can see indications of potential problem areas. These indications can be used to give guidance to the developer on areas where additional concentration is needed, as well as areas where government test efforts should be focused, such as code walkthroughs, more comprehensive unit level testing, or stress testing. Figure 7.8-2 portrays a snapshot in time of the complexity values for all the modules in a given CSCI. While the majority of them are within the range of the rules of thumb, it can be seen that several of them have well exceeded them and should be further scrutinized through testing and analysis.

Examination of complexity trends over time can also provide useful insights, especially when combined with other metrics such as design stability, the (optional) development progress, etc. For example, late software code "patches" may cause the complexity of the patched module to exceed an acceptable limit, indicating that the design rather than the code should have been changed. It is noted that test resources are better expended on modules that have a relatively high structural complexity rather than on software that will reflect a high number of lines of code tested.

Wherever possible, the cyclomatic complexity metric should be computed for the PDL. The section on rules of thumb describes special interpretation criteria for the complexity of PDL.

There are several embedded assumptions and known weaknesses in the complexity metrics. For example, in the computation of McCabe's cyclomatic complexity, there is no differentiation between the various types of control flows. A CASE statement (which is easier to use and understand than the corresponding series of conditional statements) makes a high contribution to complexity, which is somewhat counterintuitive when one considers that the corresponding series of IF..THEN..ELSE statements would probably be more troublesome from the standpoint of testing, modification, and maintenance. Further, a million straight line instructions are judged to be as complex as a

single instruction. Additionally, the interpretation of cyclomatic complexity will be different for different higher order languages.

There are innumerable ways of defining and counting lines of code. The somewhat simple definition given in the algorithm above is intended to apply somewhat equally across the spectrum of procedural languages. It should be noted that much research is ongoing into methods and tools for counting lines of code for each language. For example, the Software Engineering Institute is in the midst of such a research effort. The definition given here should at least allow historical comparisons to be done on similar implementations.

It is also recognized that percent comment lines is a very language dependent measure. Additionally, the metric does not address the usefulness or completeness of the comments. Some self-documenting languages require fewer comments than an assembly language or a language like FORTRAN. Modern programming practices (e.g,. the use of meaningful variable names, indenting, etc.) are often more helpful than archaic programming practices with many comment lines. Additional complexity metrics for languages like Ada (e.g., to measure the degree of encapsulation) can lend additional insights into the software structure.

It is recognized that the complexity metrics are highly oriented towards procedural languages. When applied to things like artificial intelligence languages and pure object oriented languages, care must be taken in interpreting the results.

For the preceding reasons and others, the complexity metrics should be used as a group. They also should be used in conjunction with the metrics for fault profiles and depth of testing.

Research is being pursued on ways of assessing the complexity of a design before any code is built. As these metrics evolve, their use should be pursued, as it is highly desirable to limit the inherent complexity of the software while in the design phase.

It is also recognized that this metric is often not applied until late in the life cycle. It is recommended that this metric should be used as soon as practical (e.g., as code is being developed). Also, it must not be relied upon as the sole metric to judge quality of the implementation of the design.

In cases of high module cyclomatic complexity, various means exist to help identify how it may be reduced. These techniques include calculation of actual complexity and essential complexity. For further details see references.

Rules of Thumb:

The criteria presented in Table 7.8-2, which should be applied at the module level, are widely accepted as industry standards. Any value not meeting these criteria should cause concern about potential problems in the specific metric area.

Table 7.8-2

Metric	Criteria (per module)
cyclomatic complexity	<= 10
control flow	= 0
volume	<= 3200
LOC	<= 200
% comment lines	>= 60 %

For cyclomatic complexity, the suggested limit is 10 for any module. The references noted show that the error rate and number of bugs observed for modules having complexities of less than 10 is substantially lower than those with complexities greater than 10. A module with complexity greater than 10 may need to be restructured (if feasible) into several less complex ones. If the complexity is due to structures like CASE statements, the complexity can be accommodated. However, if the high complexity is due to structures like DO loops and other raw logic, serious attempts should be made to redesign or subdivide the module.

A more recent approach varies the criteria for cyclomatic complexity according to life cycle phase. During the design phase it is suggested that the value not exceed 7 for program design language to allow for expected growth to a value of 10 during implementation to code.

References:

Software System Testing and Quality Assurance, Beizer, Van Nostrand Reinhold, 1984.

"Draft Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software", IEEE Computer Society P982.2/D6, May 1988.

"Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric", Thomas J. McCabe, U.S. Department of Commerce, National Bureau of Standards, NBS Special Publication 500-99, December 1982.

"Design Complexity Metrics", T.J. McCabe & Associates, Inc., May 1988.

"A Software Reliability Study Using A Complexity Measure", Walsh, 1982.

"Measuring Effectiveness and Adequacy of System Testing", Craig, Conference Proceedings, Software Test and Validation, National Institute for Software Quality and Productivity, Inc., Sept 87.

"The Complexity Analysis Tool", ARPAD-TR-88005, Oct 88.

7.9 Metric: Breadth of Testing.

Purpose/Description:

Breadth of testing addresses the degree to which required functionality has been successfully demonstrated as well as the amount of testing that has been performed. This testing can be called "black box" testing, since one is only concerned with obtaining correct outputs as a result of prescribed inputs.

Life Cycle Application:

Begin at end of unit testing.

Algorithm/Graphical Display:

Breadth of testing consists of three different measures, each of which should be applied against the set of SRS requirements, the set of IRS requirements, and the set of UFD requirements. One measure deals with coverage and two measures deal with success. These three subelements are portrayed in the following equation:

$$\frac{\#\text{ reqts tested}}{\text{total \# reqts}} \times \frac{\#\text{ reqts passed}}{\#\text{ reqts tested}} = \frac{\#\text{ reqts passed}}{\text{total \# reqts}}$$

"test coverage" "test success" "overall success"

Breadth of testing "coverage" is computed by dividing the number of requirements (either SRS, IRS, or UFD) that have been tested (with all applicable test cases under both representative and maximum stress loads) by the total number of requirements.

Breadth of testing "test success" can be computed by dividing the number of requirements (either SRS, IRS, or UFD) that have been successfully demonstrated through testing by the number of requirements that have been tested.

Breadth of testing "overall success" is computed by dividing the number of requirements (either SRS, IRS, or UFD) that have been successfully demonstrated through testing by the total number of requirements.

All three measures of breadth of testing should be tracked throughout the development process if possible; in actuality, functional testing per CSCI is normally only done as part of FQT and system level testing. The results of each must be reported and can be simultaneously displayed either over time (similar to Figure 7.10-1 in the section on depth of testing) or over key test events (see Figure 7.9-1) with the use of stacked vertical bar graphs. In practice, given the typical duration of test events such as FQT, it is useful to portray breadth of testing over time within the test so that growth of demonstrated functionality can be assessed. Figure 7.9-1 summarizes breadth

of testing over two typical test periods; in actuality, the monthly growth portrayal should also be generated for each test. It should be pointed out that test success, as defined in the algorithm above, is not computed against the same population of requirements as test coverage and overall success.

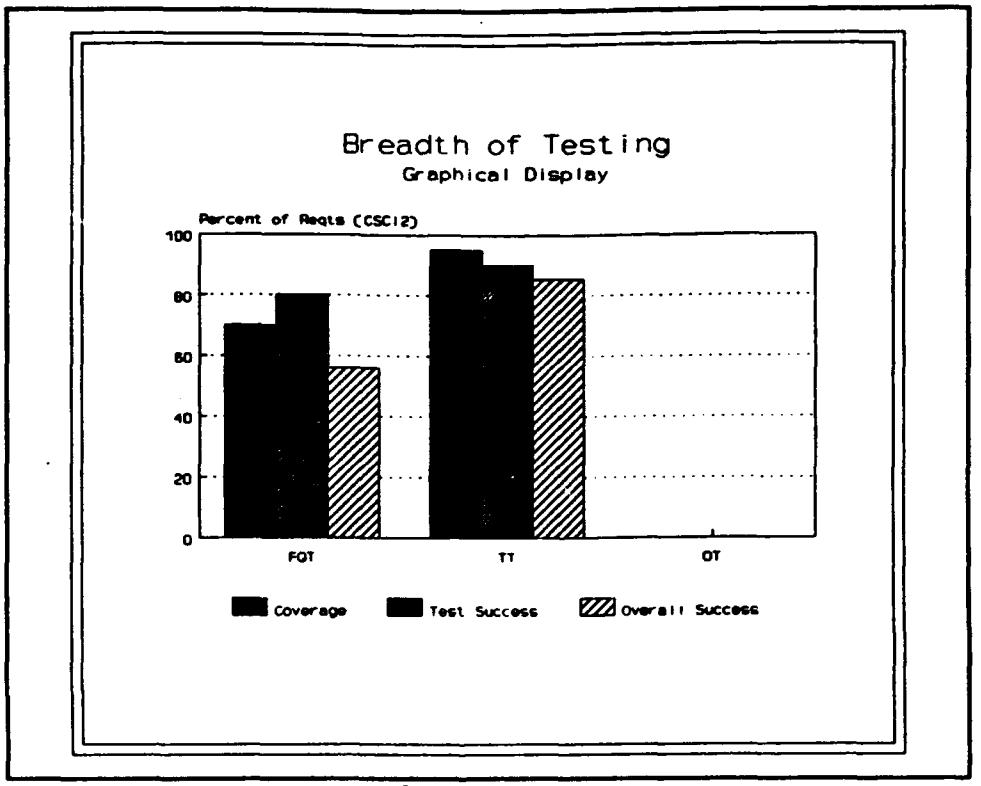

Figure 7.9-1

It should be noted that any time there is a change in requirements, breadth of testing must be revisited. A new requirements baseline will require a recomputation of the breadth of testing metrics.

Data Requirements:

number of SRS requirements
number of SRS requirements tested with all planned test cases
number of SRS requirements successfully demonstrated through testing
number of IRS requirements
number of IRS requirements tested with all planned test cases
number of IRS requirements successfully demonstrated through testing
for each of four UFD priority levels:
  number of UFD requirements

number of UFD requirements tested with all planned test cases
number of UFD requirements successfully demonstrated through testing
test identification (e.g., FQT, TT, OT)

Frequency of Reporting:

monthly throughout functionality and system level testing

Use/Interpretation:

The coverage portion of breadth of testing provides indications of the amount of testing performed without regard to success. By observing the trend of coverage over time, one gets an idea of the extent of full testing that has been performed.

The success portions of breadth of testing provide indications about requirements that have been successfully demonstrated. The test success measure will indicate the relative success based only on the requirements that have been tested. By observing the trend of the overall success portion of breadth of testing, one can easily see the growth in successfully demonstrated functionality.

One of the most innovative aspects of the UFD is the categorization of requirements in terms of four criticality levels. This approach to user requirements provides a cornerstone on which to implement the important TQM tenet of addressing customer satisfaction, and will foster a useful dialogue between the user and developer as the test process is progressing. With this approach, the most important requirements can be highlighted. Using this prioritization scheme, one should partition the breadth of testing metric to address each criticality level. At various points along the development path, the pivotal requirements for that phase can be addressed in terms of tracing, test coverage, and test success.

The test cases to be used for evaluating the success of a requirement should be developed through the TIWG process in order that sufficient test cases are generated to adequately demonstrate the requirements. This highlights the importance of the concurrent participation of the user, developer, tester, and evaluator in requirements analysis and test planning. A strong TIWG process and a healthy amount of user and tester involvement in test planning will serve to mitigate the subjectivity in assessing whether the requirements have been satisfied.

It is recognized that under the above method, failing only one test case results in a requirement not being successfully demonstrated. If sufficient resources exist, an additional, optional way to address breadth of testing consists of examining each requirement in terms of the percent of test cases that have been performed and passed. In this way, partial credit for testing a requirement can be shown (assuming multiple test cases exist for a requirement), as opposed to the all or nothing approach. This method, which is not

mandated here due to cost and reporting considerations, may be useful in providing additional granularity into breadth of testing.

The breadth of testing metrics for coverage and overall success should be used together and in conjunction with the requirements traceability metrics (to trace unsuccessfully demonstrated UFD requirements to the appropriate CSCI, and to see the test coverage on successfully traced requirements), the (optional) development progress metrics, and fault profiles so that problem areas can be isolated. The breadth of testing metric should also be used in conjunction with the metrics for depth of testing, requirements stability, and design stability.

During post deployment software support, this metric should be used with the requirements traceability metrics to indicate areas which need regression testing.

It should be noted that some requirements may not be testable until very late in the testing process (if at all). For example, a requirement that is allocated to multiple CSCIs may not be proven out until the final system level testing.

Rules of Thumb:

The government should clearly specify what functionality should be in place at each phase of development. This process is obviously system dependent. At each stage of testing (unit through system level stress testing), emphasis should be placed on demonstrating that a high percentage of the functionality needed for that stage of testing is achieved. Prior to the formal government operational test, most functions should be demonstrated under stress loading conditions.

References:

"Standard Set of Useful Software Metrics Is Urgently Needed", Fletcher J. Buckley, Computer, July 1989.

7.10 Metric: Depth of Testing.

Purpose/Description:

The depth of testing metrics provide indications of the extent and success of testing from the point of view of coverage of possible paths/conditions within the software. The testing can be called "white box" testing, since there is visibility into the paths/conditions within the software.

Life Cycle Application:

Begin at CDR. Continue through development as changes occur in either design, implementation, or testing. Revisit as necessary during PDSS.

Algorithm/Graphical Display:

Depth of testing consists of three separate measures, each of which is comprised of one coverage and two success subelements (similar to breadth of testing).

The path metric for each module is defined as the number of unique paths in the module that have been successfully executed at least once, divided by the total number of paths in the module. A path is defined as a logical traversal of a module, from an entry point to an exit point. If one refers back to the discussion of the complexity metric, a path is actually a combination of edges and nodes.

The statement metric for each module is the number of executable statements in the module that have been successfully exercised at least once, divided by the total number of executable statements in the module.

The domain metric for each module is the number of input instances that have been successfully tested with at least one legal entry and one illegal entry in every field of every input parameter, divided by the total number of input instances in the module.

The optional decision point metric for each module is the number of decision points in the module that have been successfully exercised with all classes of legal conditions as well as one illegal condition (if any exist) at least once, divided by the total number of decision points. Each decision point containing an "or" should be tested at least once for each of the condition's logical predicates.

Figure 7.10-1 portrays test coverage and overall success for the paths within a CSCI, which is computed as a composite of its module level path metrics. The graph is somewhat similar to the graph portrayed for breadth of testing, although the depth of testing metric will typically progress much quicker than the breadth of testing metric.

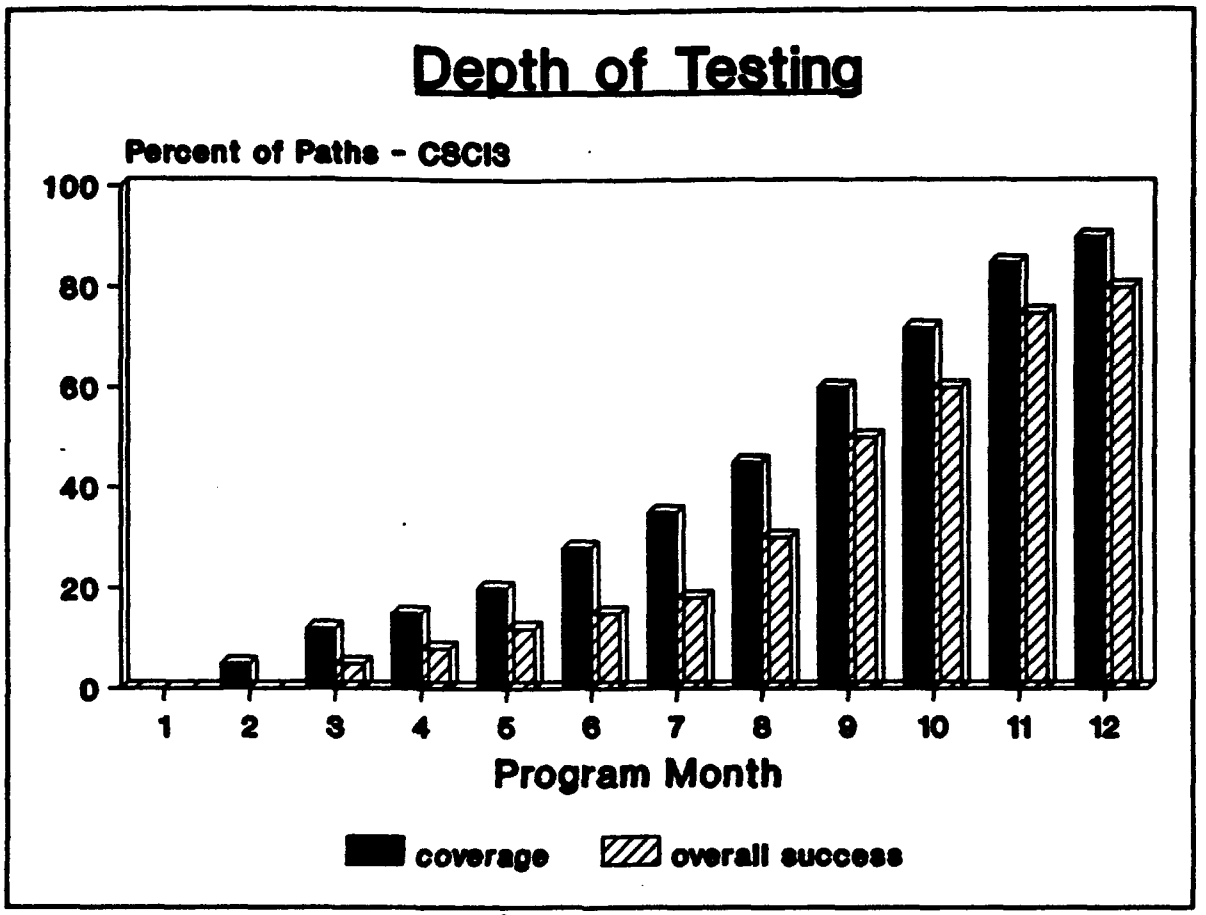

Figure 7.10-1

Data Requirements:

for each module:
- CSCI identifier
- CSC identifier
- CSU identifier
- number of paths which have been successfully executed at least once
- number of inputs which have been successfully tested with one legal entry and one illegal entry
- number of statements that have been successfully exercised
- number of paths
- number of statements
- number of input instances
- number of decision points (optional)

Frequency of Reporting:

monthly (if a CSU has been modified or further tested)

Note: In recognition of the effort required to collect and report the depth of testing metric, the following rules are offered for data collection

and reporting. Compute the domain metric always (it is relatively straightforward). For the path and statement metrics, if automated tools exist, compute the metrics. If no automated tools exist, compute the metrics if the module implements a UFD priority one requirement, or if the complexity rules of thumb are exceeded for that module.

Use/Interpretation:

The depth of testing metric attacks the issue of test coverage, test success, and overall success by considering the paths, statements, and inputs to the software. The elements of test coverage, test success, and overall success should be used and interpreted in a similar fashion to that described in the use/interpretation section of the breadth of testing metric. The trends of these depth of testing metrics over time provide indications of the progress of successful testing, and also of the sufficiency of the testing.

The metrics are collected at the module or CSU level, but they can be easily extended to the CSC, CSCI, or system level by simply replacing the term "module" in the algorithm definition above with the term "CSC," or "CSCI," or "system." Early in the contractor testing process, it makes more sense to assess depth of testing at the module or CSU level, but later it makes more sense to consider CSCs and CSCIs.

The depth of testing metrics are typically collected using a combination of static and dynamic analysis techniques. Static analysis deals with examinations of the software and related documentation; an example of such a tool would be a complexity analysis tool used to compute the possible paths through the software. Dynamic analysis deals with actually executing the software; an example would be a profiler that tracks the execution of various paths within the software. Test case generators can be used to aid in the development of appropriate test cases. Tools are available to help in the data collection effort; they serve to greatly ease the dynamic analysis effort.

The depth of testing metrics can be used to focus discussion on those modules which implement high priority UFD requirements. Ideally, the test process should initially focus on those modules which incorporate high priority UFD requirements. One would like to see these modules, as well as common or shared modules, "wrung out" early in the test program via the depth of testing metric.

In addition to addressing test coverage and test success, the domain metric and the (optional) decision point metric serve to partially address the robustness of the software. That is, these metrics serve to indicate the extent to which the software can withstand the entry of illegal input values.

The depth of testing metrics should be used in conjunction with requirements traceability, fault profiles, complexity, and the (optional) development progress. For example, with complexity, the modules of highest complexity could be highlighted for testing (e.g., one might want to test those modules first). Also, the cyclomatic complexity value gives the required number of test cases needed to exercise all the paths in the software. They must also

be used with the breadth of testing metrics to insure that all aspects of testing are approaching an acceptable state for the government.

Rules of Thumb:

None.

References:

"Standard Set of Useful Software Metrics Is Urgently Needed", Fletcher J. Buckley, Computer, July 1989.

7.11 Metric: Fault Profiles.

Purpose/Description:

Fault profiles provide insight into the quality of the software, as well as the developer's ability to fix known faults. It is interesting to note that these insights actually come from measuring the lack of quality ("faults") in the software. Early in the development process, fault profiles can be used to measure the quality of the translation of the software requirements into the design. Later in the development process, they can be used to measure the quality of the implementation of the software requirements and design into code.

Life Cycle Application:

Begin after completion of unit testing (when software has been brought under developer's configuration control) and continue through PDSS.

Algorithm/Graphical Display:

Plot cumulative number of detected software faults and cumulative number of closed software faults as a function of time, as shown in Figure 7.11-1. One plot should be developed for each CSCI and for each priority level, as defined in the data requirements section below.

One should also plot, on a month by month basis, the number of software faults that were detected and closed during the month, as shown in Figure 7.11-2.

Calculate average age of closed faults as follows: For all closed STRs, sum the days between when the STR was opened and the STR was closed. Divide this sum by the total number of closed STRs.

Calculate average age of STRs as follows: For all STRs, sum the days between when the fault was opened and either when it was closed or the current date (if still open). Divide this sum by the total number of STRs.

Relative CSCI status with respect to open faults can be shown as in Figure 7.11-3. Histograms of open faults by CSCI and priority can be portrayed as in Figure 7.11-4.

Finally, another useful display is the average age of STRs over time, as shown in Figure 7.11-5.

An attempt should be made to screen out duplicate software problems before formally entering an STR. In cases where it is determined, subsequent to the formal entry, that an STR is a duplicate, its status should be changed to duplicate, even though the original problem may not be resolved. When changing an STR from an open status to a duplicate status, the cumulative number opened should be decremented for the next reporting period. The original STR remains open until a fix is developed and verified. It should be recognized that an extensive number of open STRs that will turn out to be duplicate may skew the parameters computed above.

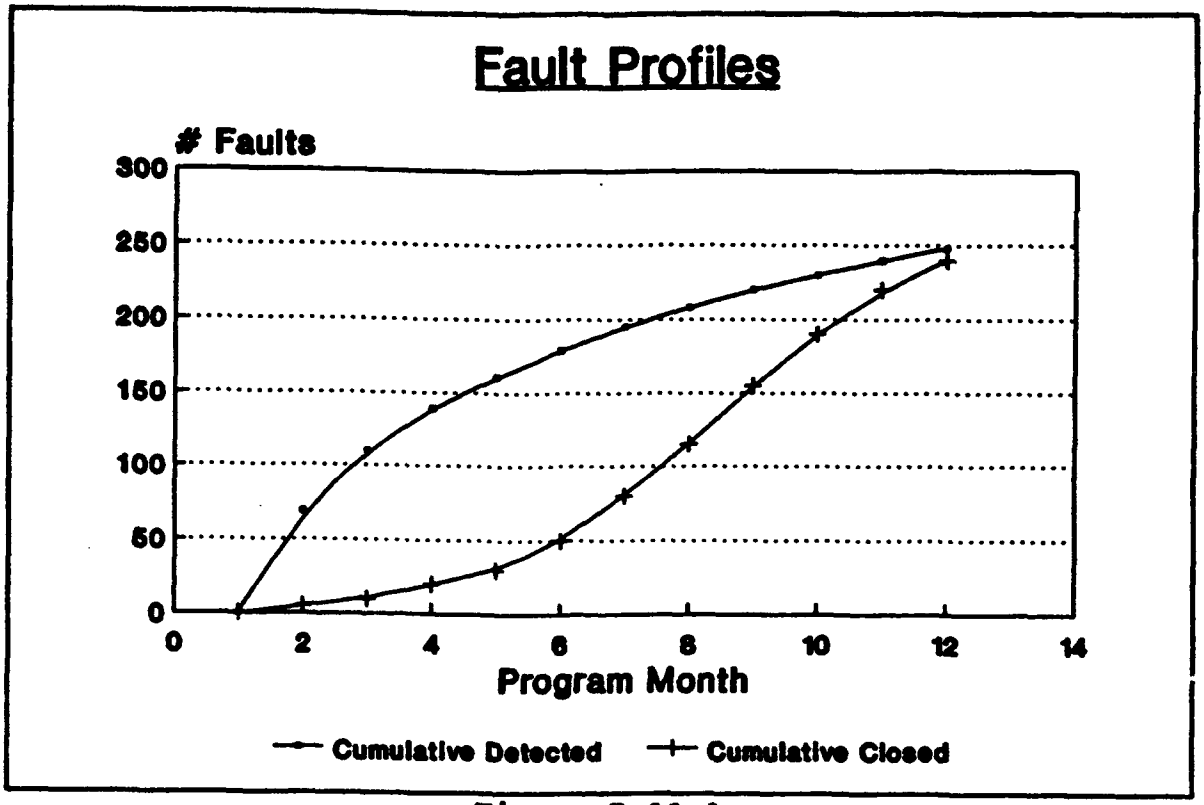

Figure 7.11-1

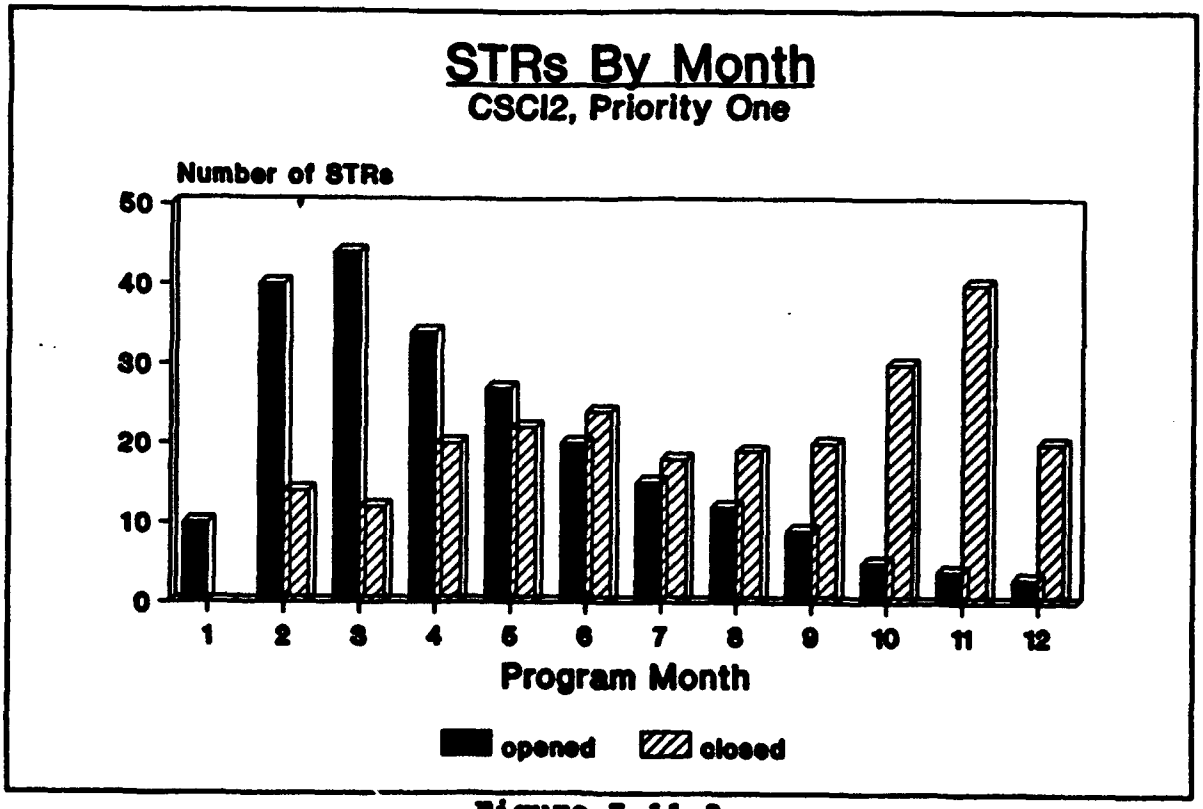

Figure 7.11-2

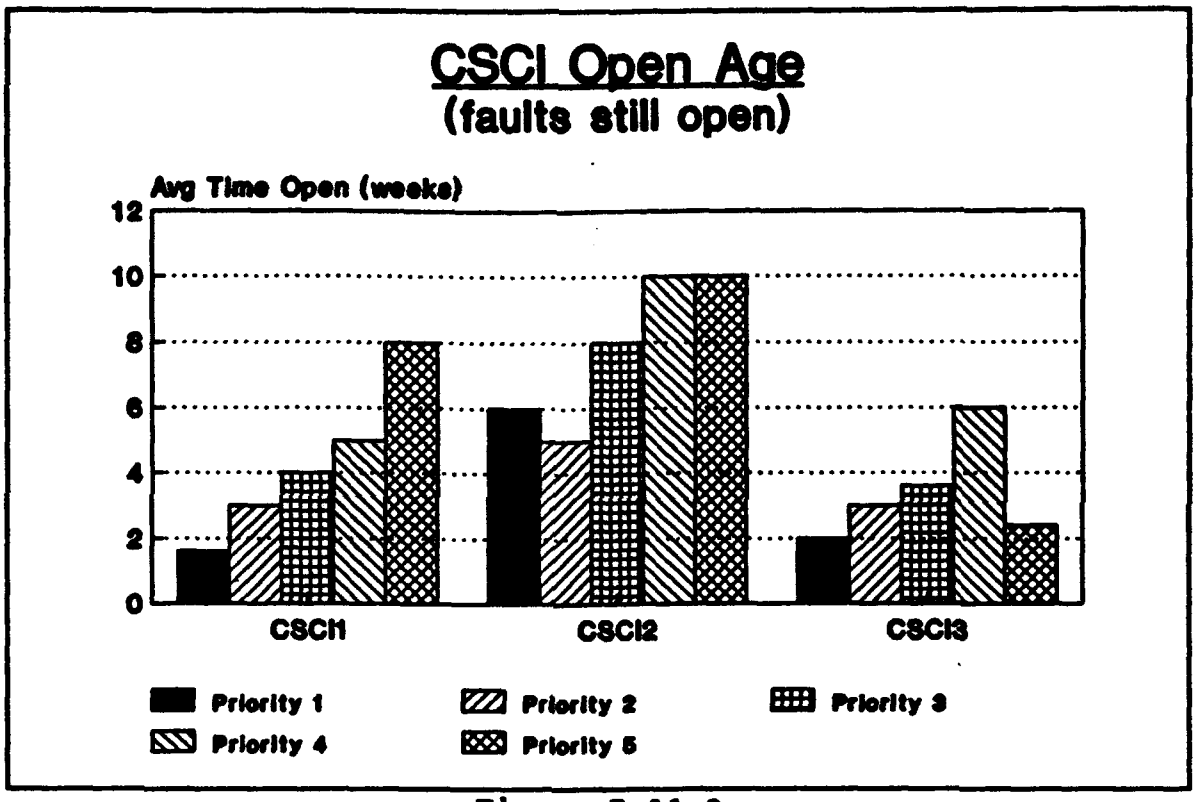

Figure 7.11-3

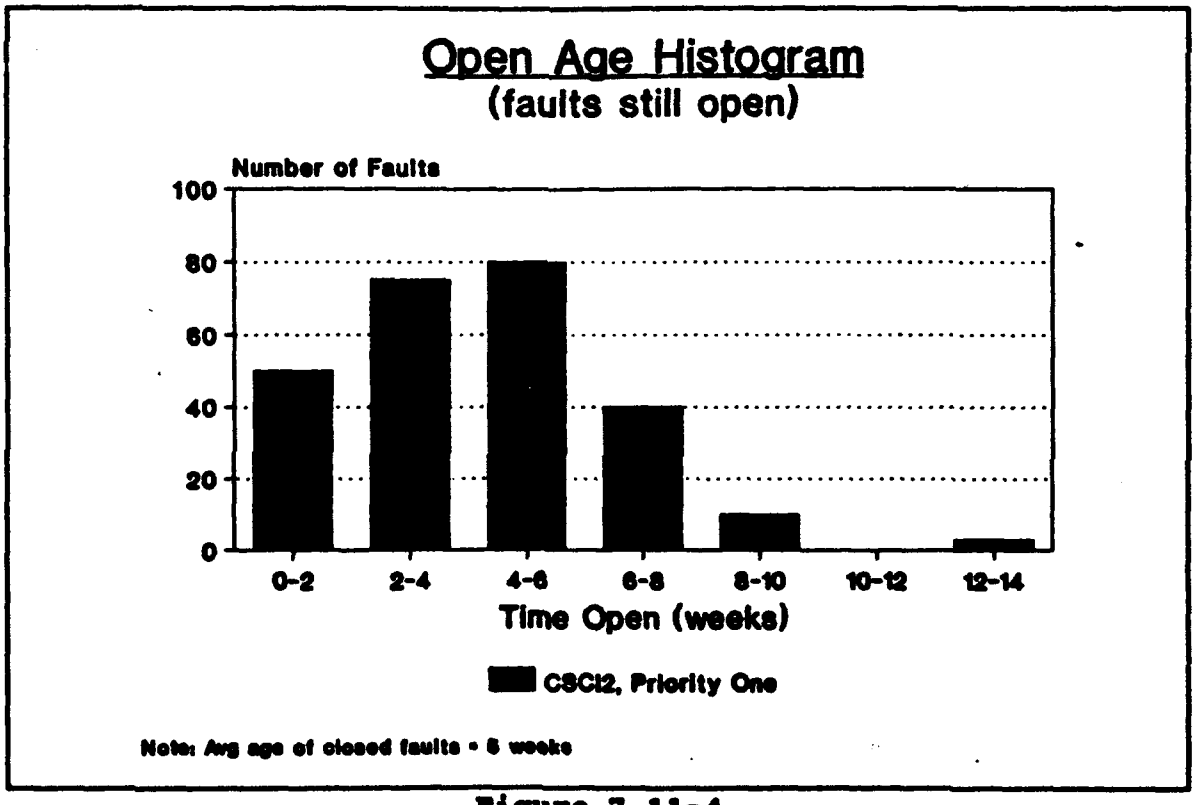

Figure 7.11-4

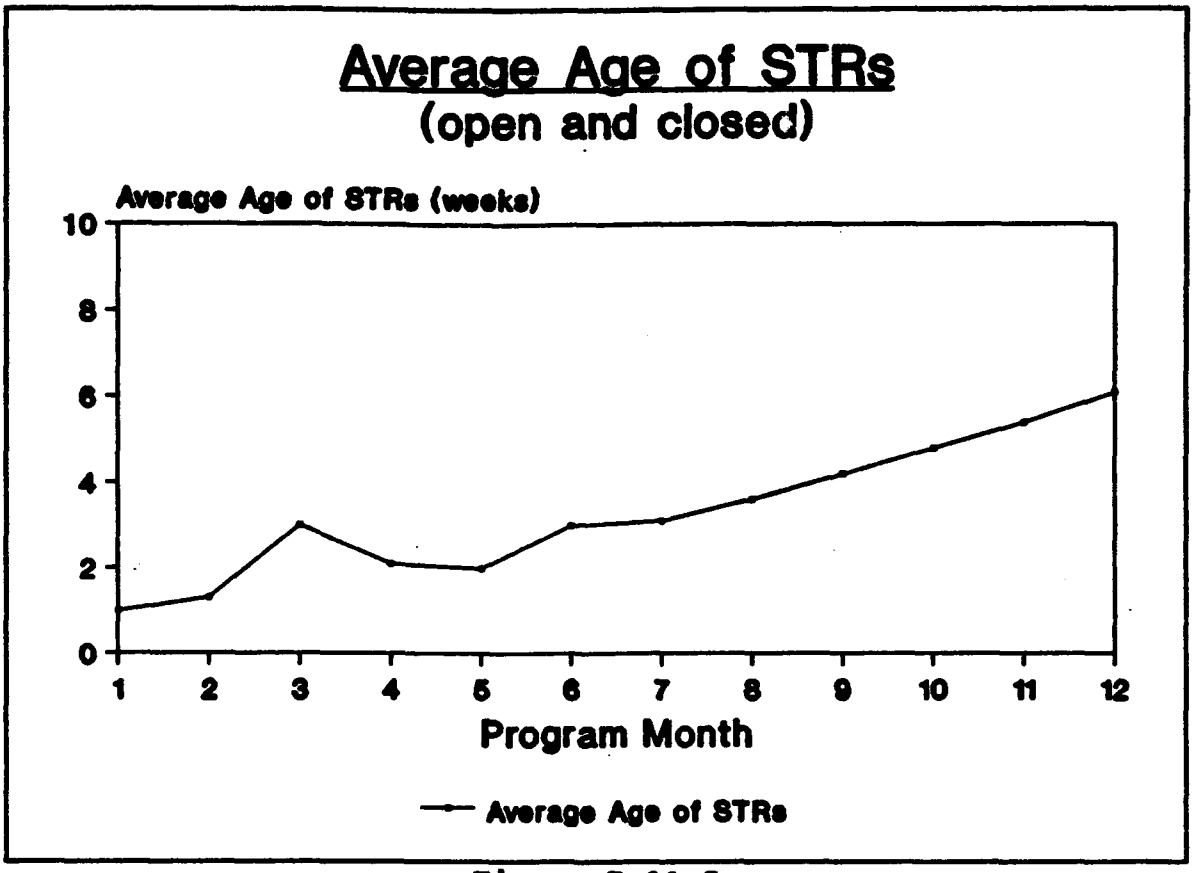

Figure 7.11-5

Data Requirements:

The following raw data should be derived from all software trouble reports. These data will be used to calculate higher level statistics (described later in this section), as well as to support queries generated as a result of the graphical portrayals.

unique id
date written
descriptive title of problem
detailed description of problem (optional)
priority :
- 1 = causes mission essential function (or operator's accomplishment thereof) to be disabled or jeopardizes personnel safety
- 2 = causes mission essential function (or operator's accomplishment thereof) for which there is no work around to be degraded
- 3 = causes mission essential function (or operator's accomplishment thereof) for which there is a reasonable work around to be degraded
- 4 = causes operator inconvenience but doesn't affect a mission essential function
- 5 = all other errors

category :
requirements
design
code
documentation
other
when discovered :
requirements analysis
design review
code and unit test
integration and test
operation/maintenance
status :
open
duplicate
closed
invalid
date detected
date closed
software module
CSCI
software version
effort to fix (man-hours)

The following rolled up statistics are the building blocks for the graphical portrayals of fault profiles.

For each priority, for each CSCI :
cumulative number of STRs
cumulative number of closed STRs
average age of closed STRs
average age of STRs
total number for each category (described above)

Frequency of Reporting:

monthly

Use/Interpretation:

There are various aspects of fault profiles that can be examined for insights into quality problems. The most popular type of graphical representation, portrayed above in Figure 7.11-1, displays detected faults and closed (corrected and verified) faults on the same scale. These types of graphs should be examined for each priority level, and for each major module or CSCI. Applied during the early stages of development, fault profiles measure the quality of the translation of the software requirements into the design. STRs opened during this phase suggest that requirements are not being defined or interpreted correctly. Applied later in the development process, assuming adequate testing, fault profiles measure the implementation of the requirements and design into code. STRs opened during this stage could be the result of having an inadequate design to implement the requirements, or a poor

implementation of the design into code. An examination of the fault category should provide indications of these causal relationships; these examinations should be performed as a matter of course in any analysis of fault profiles. One should continuously observe the gap between open and closed faults; if a constant gap or a continuing divergence is observed, especially as a key test or milestone is being approached, appropriate action should be taken. The only time the open curve should decrease is when duplicate STRs have been discovered subsequent to entry, and the number open has been decremented as described in the algorithm above.

Another use of fault profiles consists of monthly non-cumulative totals for each CSCI and priority (Figure 7.11-2). This can be compared to the amount of testing done in those months to provide insights into the adequacy of the test program.

Open age histograms (Figure 7.11-3 and Figure 7.11-4) can be used to indicate which CSCIs, which priorities, and which faults are the most troublesome. This may serve to indicate that the developing group for that CSCI may need assistance, whether due to a difficult set of requirements or for some other cause.

Average open age graphs (Figure 7.11-5) can track whether the open age of faults is increasing with time, which may be an indication that the developer is becoming saturated or that some faults are exceedingly difficult to fix.

Caution must be used in interpreting the fault profiles, as the detection of errors is closely tied to the quality of the development and testing process. That is, a low number of detected faults could indicate a good product from a good process or simply a bad process to start with (e.g., one with inadequate testing). Conversely, a large number of faults early on in a program may not be bad (e.g., the developer may have an aggressive test program, or may be using techniques such as rapid prototyping with heavy user involvement to wring out the requirements early). A large number of STRs open in a particular month may be the result of errors detected during a specification review, audit, test, or from use of the software in the field. Thus, the measures cannot be assessed without also considering the measures on breadth and depth of testing. The fault profiles should also be used in conjunction with the metrics for complexity, design stability, and requirements stability.

If the cumulative number of closed STRs remains constant over time and a number of STRs remain open, this may indicate a lack of problem resolution. The age of the open STRs should be checked to see if they have been open for an unreasonable period of time. If so, these STRs represent areas of increased risk. The cause for lack of resolution needs to be identified and corrective action taken.

Once an average STR age has been established, large individual deviations should be investigated. There are several reasons why STRs may remain open for a lengthy period of time. One could be that the STR is a result of identification of an inadequate requirement which needs to be refined and is undergoing review. An ECP-S may have been written for a problem noted and is

waiting resolution. It could also mean that the developer has failed to take corrective action on the problem. Again, the reasons for lack of problem resolution need to be identified and corrective action taken. The average open age of high priority faults should also be examined with respect to the time remaining to the next major test or milestone. If the average open age of high priority faults exceeds the time remaining, consideration should be given to delaying the test or milestone until the problems are resolved.

As an option the following method of assessing test adequacy can be used. Rome Air Development Center (RADC) conducted wide research on several software projects and identified the characteristics which have direct impact on system reliability. RADC-TR-87-171 Volumes I and II, "Methodology for Software Reliability Prediction" can be used to predict the number of faults expected to be present per configuration item. Using this information as a guideline, fault profiles can be compared to these estimates to determine if the "peak level" of opened STRs is being approached and that the software development process has matured. As an example, suppose the predicted number of faults is much higher than the actual number of faults reported. If the test coverage metrics are low, this suggests that testing is not complete and the remaining faults are yet to be found. If the test coverage is high this could mean that the software was well written to start with, or that the test cases used were not thorough enough. If the prediction is much less than the actual number of faults, this may indicate a number of problems. The software developer may have an inadequate development effort, may have encountered unexpected design difficulties, faulty coding or an especially troublesome module(s). The requirements stability metric should be checked; if it is high, this may indicate an immature baseline.

Rules of Thumb:

The government should not accept software for any formal system level government testing until, at a minimum, all priority one and two faults have been closed. Furthermore, a large number of lower priority faults should be examined for a possible cumulative effect on successful test conduct.

If tracking the fault profiles starts early in software development, an average STR open age of less than three months may be experienced. After fielding this value can rise, primarily due to the necessary delays typically experienced in the PDSS process.

References:

AMC Pamphlet 70-14, "Software Quality Indicators", 20 Jan 87.
SEI Quality Subgroup Working Papers, untitled, Nov 1989.
IEEE Standard 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software.
IEEE Standard 982.2-1988, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

7.12 Metric: Reliability.

Purpose/Description:

The reliability metric is an indicator of how many faults there are in the software as well as the number of faults expected when the software is used in its intended environment. An additional measure, Time To Restore, augments reliability by measuring the impact of unexpected outages.

Life Cycle Application:

Begin at CDR.

Algorithm/Graphical Display:

Use fault profile metrics throughout development. After turnover to the government for TT and OT, calculate the system Mean Time Between Hardware/Software Mission Failure (MTB HW/SW MF) and Mean Time Between Software Mission Failure (MTB SW MF) of the latest configuration directly only when the software is being used and stressed in accordance with its Operational Mode Summary/Mission Profile (OMS/MP). The nature of TT sometimes dictates that the OMS/MP can not be adhered to. In these cases there are frequently sufficient data available on each of the system mission essential functions which comprise the OMS/MP to normalize the observed function failure rates to the usage proportions described in the OMS/MP to predict how often the software will fail in the field application. Progress can be tracked using various reliability growth techniques. Note that unlike fault profiles, wherein duplicate occurrences of the same software fault are not counted, when calculating the various MTBF parameters, every occurrence of repeat failures must be included. "Time" in this application is system operating hours, not CPU time. A useful supplementary display consists of a Pareto distribution of each software fault type showing its frequency of occurrence; Figure 7.12-1 portrays a sample Pareto plot.

Time To Restore consists of various measures of the time it takes an operator to restore mission essential automated capabilities due to failure of the software. This time includes recovery of lost data and any system reinitialization that is required to return the system to full functionality.

Due to the specialized nature of some of the above techniques, it is recommended that the values be provided by either an independent evaluator or a PM matrix support reliability analyst.

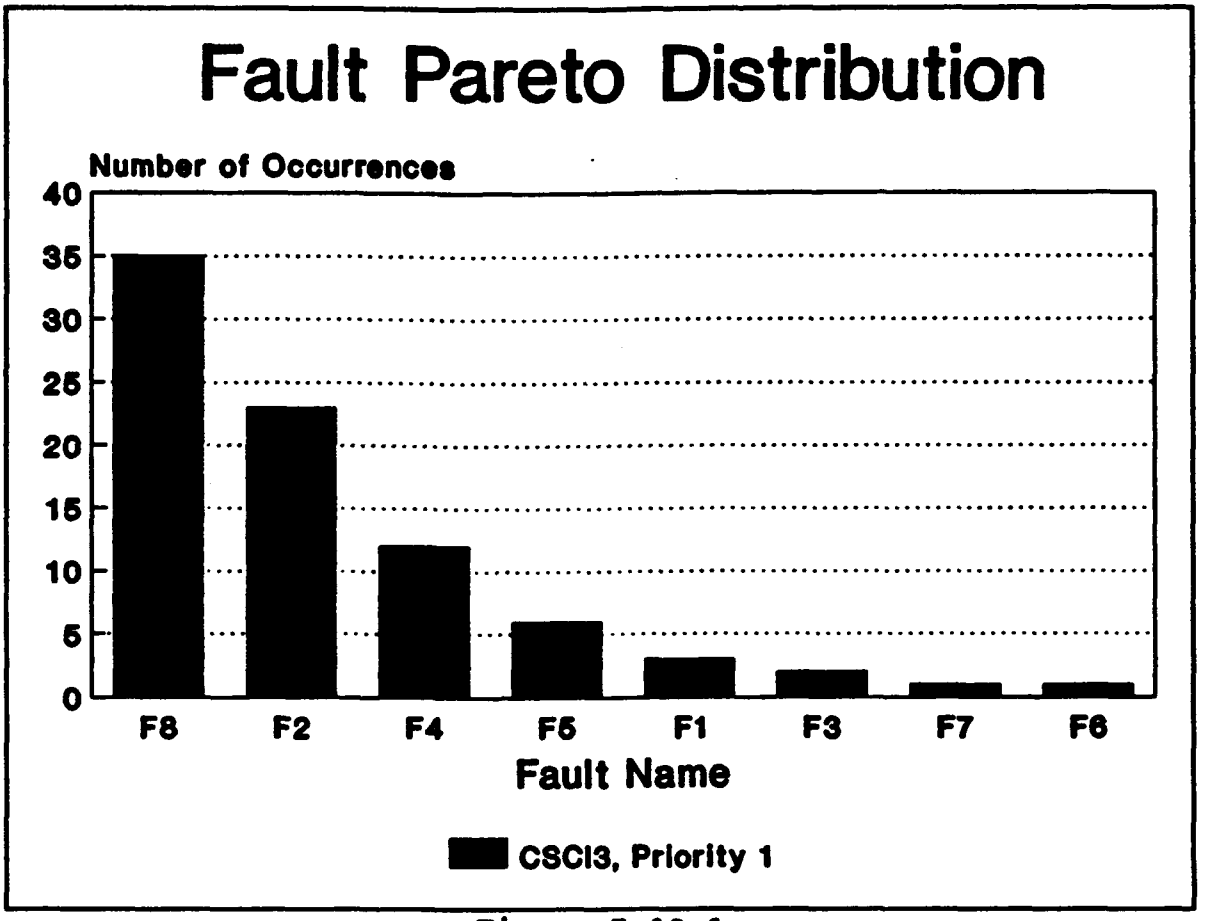

Figure 7.12-1

Data Requirements:

All data requirements from the fault profile metrics.
All reliability, availability, and maintainability incident data from test
Test identification (e.g. Pre-Production Qualification Test (PPQT), Reliability Growth Test, Independent Operational Test & Evaluation (IOTE))
System Mean Time Between Operational Mission Failure (MTBOMF) requirement
System MTB HW/SW MF requirement
System Mean Time To Repair requirement
Demonstrated point estimate and 80% lower confidence bound on MTB HW/SW MF
Demonstrated point estimate and 80% lower confidence bound on MTB SW MF
Number of software failure modes and number occurrences of each
Time To Restore (mean, median, and maximum 95th percentile)

Frequency of Reporting:
Fault profile information - monthly.

Contractor TT or Government TT/OT - gather data in real time as incidents occur.

Use/Interpretation:

Use of the fault profile metrics provides indications of the rate at which faults are being reduced and thus reliability increased. Of course, one also needs to simultaneously consider the test coverage metrics. The fault profile information, however, says nothing about how often the faults remaining in the software will be encountered by the user. While many arguments can be made that Mean Time Between Failure (MTBF) is an inappropriate measure, it is more than adequate for estimating how often one can expect the software to "fail" in a field environment as long as inputs are of the type and in relative proportion to what will be encountered in field use, and modules are exercised with the relative frequency expected in a tactical environment. Measuring MTBF only when the system and software are being used in accordance with the OMS/MP or normalizing mission essential function test data to the OMS/MP insures the above conditions.

The calculated 80% lower confidence bound on MTB HW/SW MF should be compared to its requirement. In cases where the demonstrated is below required, MTB SW MF allows one to determine the contributions of both hardware and software to system unreliability.

A Pareto distribution will enable the developer to focus corrective actions on faults which contribute most to software unreliability (i.e., focus on the significant few (with a high frequency of occurrence) as opposed to the trivial many).

The calculated mean and median Times To Restore should be compared to the system Mean Time To Repair requirement. Excessive mean, median, or individual times can indicate poor design or human factors problems. The three measures together give an indication of the dispersion of the individual restoration times.

Rules of Thumb:

No formal criteria are given. However, the government should not accept the software until the fault profile open anomaly curve flattens out, closure of anomalies approaches the open anomalies, and a high degree of breadth and depth of testing have been demonstrated. Do not go to OT until system level MTB HW/SW MF has either been demonstrated with high confidence in TT or is projected to exceed the requirement based on late occurring corrective actions.

References:

AR 702-3, "Army Materiel Systems Reliability, Availability and Maintainability"

TRADOC/AMC Pamphlet 70-11, "Reliability, Availability, and Maintainability (RAM) Rationale Report Handbook"

8. IMPLEMENTATION CONSIDERATIONS

8.1 Qualifying Rules.

The suite of metrics must be used on all systems that meet all three of the following criteria:

1) the system contains computer hardware and software developed and managed under the auspices of either of the following regulations:
   a) AR 70-1, "System Acquisition Policy and Procedures"; or
   b) AR 25-3, "Army Life Cycle Management of Information Systems"
2) the system is either :
   a) an MSCR acquisition category (ACAT) I, II, or III; or
   b) an AIS Class 1, 2, 3, or 4.
3) the system either :
   a) supports decision making; or
   b) receives or senses analog or digital data, processes it and uses it to influence external system behavior, including input from soldiers such as crew controls; or
   c) exchanges data with other systems; or
   d) stores information for access by soldiers (e.g., target priorities, digitized maps)

8.2 Grandfathering.

Headquarters, Department of the Army has directed that, beginning on 1 June 1992, unless the decision body (ASARC or MAISRC) directs otherwise, the following grandfathering rules are to be used to decide if an existing program must implement the metrics set:

1) for qualifying (see above) MSCR systems, the full set of metrics is required for all systems which are not past MS II.
2) for qualifying (see above) AIS systems, the full set of metrics is required for all systems which are not past MS III.C (certified effective and suitable for full fielding).

Note: major product improvement activities (in the post-deployment phase for MSCR systems, or in the operations phase for AIS systems) which meet the criteria above will be required to implement the full set of metrics.

8.3 Data Collection.

For most metrics, the data should continue to be collected after fielding of the system. In particular, software updates developed through a Life Cycle Software Support Center (LCSSC) must be measured in a similar fashion as a developmental system. The general rule is to collect the same basic data that were collected in similar activities prior to Milestone III.

When scanning across all metric data elements, one can observe that some subelements are identical for different metrics. When reporting the metrics to the data base (described in Section 12), a smart interface will insulate the data entry person from having to enter the same information more than once.

The Program Manager has responsibility for seeing that the metric data elements are collected. In reality, the actual collection will most probably be done by the developer or IV&V agent. In those cases where the developer is doing the collection, it is recommended that the IV&V agent should at least sample some of the more crucial metric data elements for consistency and correctness.

As the metrics are reported on the periodic basis described herein (monthly for many metrics), it is entirely possible that certain data elements may not have changed from the previous reporting period. In these cases, it will be acceptable for the data entry person to simply report "no change" for the particular data element.

To facilitate the collection of the basic data elements, data item descriptions (DIDs) are being developed. The master set of DIDs will be based upon the data requirements prescribed for each metric in Section 7. Additionally, a subset of these data requirements will comprise a second set of DIDs, which will support collection for the metrics data base described in Section 12.

8.4 Life Cycle Application.

Figure 8-1 portrays a graphical depiction of the applicability of metrics throughout the life cycle. Both MSCR and AIS milestones are listed in the top portion of the chart. Although a waterfall model is portrayed in the activities line at the top of the chart, the chart can also be applied to other life cycle models (e.g., the spiral model) by using the specific activities identified at the top of the chart as cues to the use of a specific metric. As is indicated on the chart, development activities during the post-deployment software support phase require the collection of similar metrics as would be collected on the same activities during a pre-deployment phase.

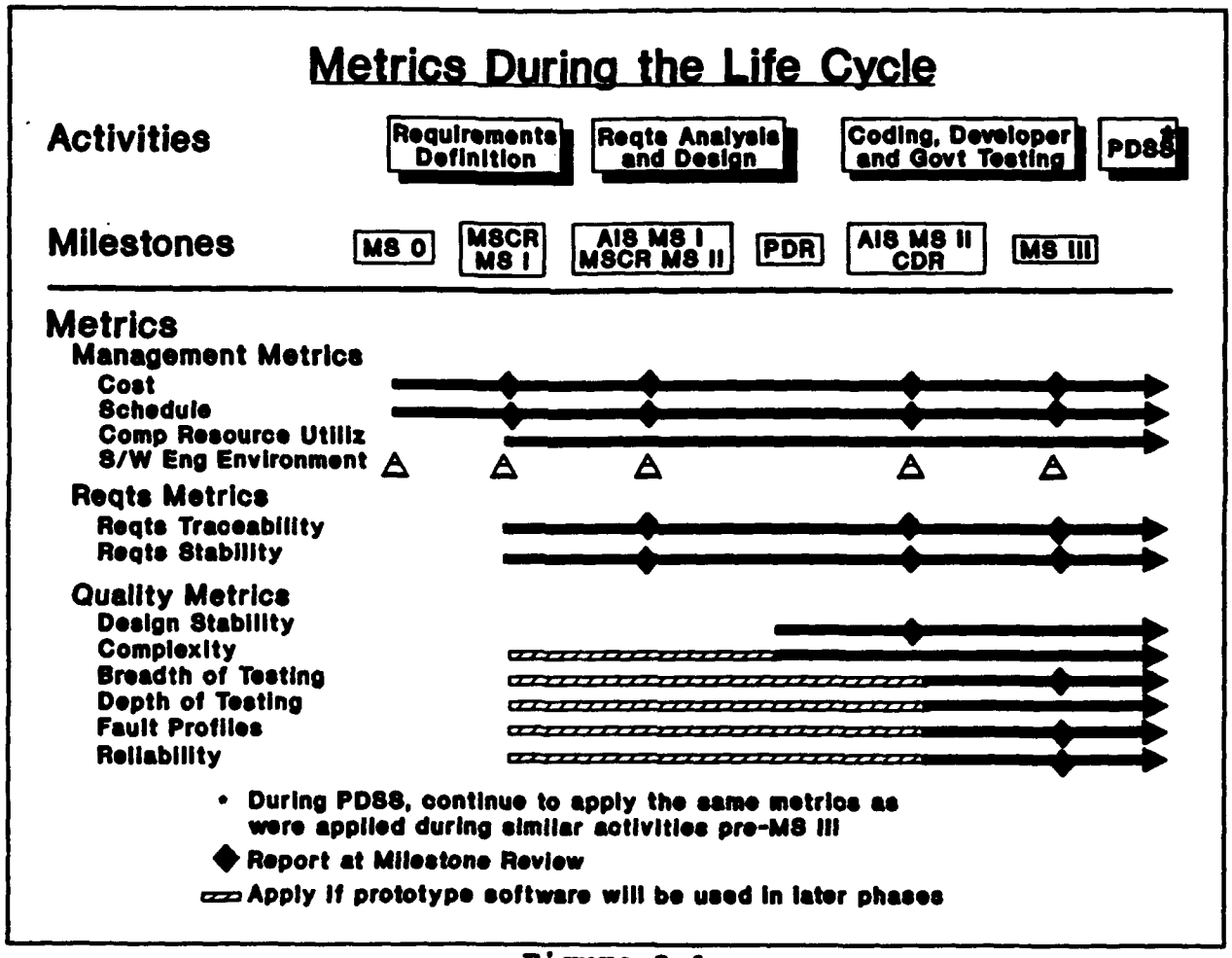

Figure 8-1

The twelve chosen metrics are felt to provide valuable insight into a program, especially with regard to demonstrated results and readiness for test. As such, all should be reported at Program Reviews, In Process Reviews, and Test Readiness Reviews (both Technical and Operational). Some of the more detailed metrics and many subelements of the major ones however, are not suitable for presentation at high level decision reviews such as ASARCs, MAISRCs, and Defense Acquisition Boards (DABs). Within Figure 8-1, the diamonds indicate the major decision milestones at which the given metrics should be reported. In addition to the metrics considered mandatory for a given milestone, any other metric which indicates the potential for serious problems should be reported.

9. TAILORING

The subject of tailoring of the metric set was exhaustively discussed during the proceedings of the Measures Subgroup. While there is a school of thought that believes that the program manager should be allowed to tailor the metric set to the program under consideration, the vast majority of STEP participants believe that the only way to implement a metrics program throughout the Army is to require the collection of a common set of metric data elements.

Tailoring above and beyond the core set of twelve metrics is certainly encouraged. For example, the appendices describe two optional metrics that some groups have found beneficial to use in the past. For a particular program, the use of any other measure deemed appropriate by the materiel developer (or by the test and evaluation community as negotiated through the TIWG) is also encouraged. For example, a particular functional area within the Army may believe that function points as a metric makes sense for their applications. Another way of tailoring would be to take the basic data elements contained in this report and put them together in different ways. These new ways of combining existing data elements might be useful as an adjunct to the required methods described herein.

There are also three ways to tailor metrics within the core set. First, one could report the metrics more frequently than is called for in this document. For example, if a metric is experiencing turbulence, it may be desirable to obtain more frequent measurements of progress. The second way to tailor within the metric set is from the standpoint of depth. It is certainly allowable to go down to further levels of depth, in addition to the level described for each metric. For example, in the cost metric, it may be beneficial to go down deeper in the work breakdown structure, to gain additional visibility into cost considerations. The third way to tailor within the metric set is from the standpoint of level of resolution. It is allowable to go down to a finer resolution, in addition to what is described herein. For example, where we track a metric by CSCI, it may be desirable in certain instances to also track the metric by that CSCI's constituent parts.

10. JUSTIFICATION FOR METRIC SET

The Measures Subgroup feels that all the metrics specified in the metric set meet the criteria set forth in Section 5. In addition, they are consistent with the tenets of total quality management in that they represent both process and product measures. They also address the areas most commonly agreed to as needing attention in order to develop "good" software: requirements, resources, development process, quality/maturity, and test coverage/sufficiency. Table 12-1 illustrates the support that the metrics lend to the overall software quality program described in DOD-STD-2167A.

The challenge in software development is to make informed, fact-based management decisions. The metrics serve to act as an early warning system, highlighting small problems early so that they may be resolved before they grow into larger, more costly problems.

It should be noted that the metrics selected may not be the best set. To date, none of the metrics in this report have been formally validated in the sense of scientific experiments showing direct correlation to program success or failure. However, many of the metrics have been used successfully on Army and other Service programs. For example, cyclomatic complexity was used successfully on several programs, including the Howitzer Improvement Program (HIP) and Backup Computer System. Fault profiles are being used widely and were beneficial on HIP, Conduct of Fire Trainer and the All Source Analysis System. At the current time, these metrics comprise what is thought to be a logical starting point. The philosophy employed by the group was to move beyond theoretical discussions (which are important but best left to the academic world) and arguments about relative strengths and weaknesses of each metric and get on with the process of measuring software.

A period of use and an eventual formal, scientific validation must follow the mandate to use this set. Lessons learned from that period will be used to improve the metric set. This process should be iterative, and as such should be considered both a short term and long term effort.

Table 12-1 Metrics Support to Software Quality

Metric	DOD-STD-2167A Software Quality Factors										
	Correctness	Usability	Rel	Efficiency	Integrity	Testability	Maintainability	Flexibility	Portability	Reusability	Interop
Cost											
Schedule											
CRU				P				S	S		
SEE	S	S	S	S	S	S	S	S	S	S	S
Reqts Traceability	P				S	P	S	S			
Reqts Stability	P										
Design Stability	S										
Complexity			S	S		P	P	P		S	
Breadth of Testing	P		P		S	S				P	
Depth of Testing	P		P		P	S	S			P	
Fault Profiles	P		P		S					P	
Reliability	S		P							P	

P = primary influence
S = secondary or indirect influence

11. COSTS AND BENEFITS

Implementation of a structured approach to collect metric data may increase the cost of software development, although as the process maturity of various developers increases, the cost of a metrics program will decrease. It might also be argued (by someone with a very narrow perspective) that the activity of measurement will reduce productivity on a project. However, while it is true that implementation of a metrics program will add some cost to a program, given the amount of money the Army spends on software development and the fact that there seems to be a systemic problem in the timely delivery of quality software, there is certainly a potential for cost benefit or cost avoidance (although not measurable at the current time), especially in the long run.

Experience on the cost of collecting metric data is sketchy. The most definitive information obtained came from NASA's Software Engineering Laboratory. While this is a somewhat different environment for software development from that of the Army, NASA has over fifteen years of experience in the collection and analysis of metric data. Their experience suggests that data collection costs about 2 to 3 percent of the cost of software development. Data entry and quality control comprises another 6 to 8 percent of the cost, and analysis adds another 10 to 15 percent to the cost. It is felt that the last category (analysis) would not be an added burden to Army software development costs, since the evaluation agencies would absorb this burden.

Also, it cannot be stated that there is evidence that metrics "pay off," at least in the monetary sense but it makes logical sense that they would. It is felt at this point that measurement of software is not only good to do but in fact necessary. It enables management and decision makers to understand and gain visibility into the software development process. Also, these same people need basic data to manage and predict future projects. The state of the art cannot be advanced without measurement, applied research, and analysis.

12. METRICS DATA BASE

During the recommendations phase of the STEP, the Measures Subgroup proposed that a centralized database be created to store the data for the minimum set of metrics to be collected throughout the Army. There were two reasons for creating the database. First, it would enable decision makers and evaluators to easily monitor a program's progress at any point in time without putting an extra burden on a PM during a time when the PM's staff might be very busy (e.g., in preparing for a milestone review). Second, and most importantly, it would greatly facilitate the stepwise refinement, formal validation, and cost benefit analyses of the metrics as data from numerous projects would be available from one source. The proposal was endorsed by the Vice Chief of Staff of the Army.

TECOM has volunteered to develop, house, and administer the database. A TECOM-led contractual effort is ongoing to finalize the metrics database requirements specification and design document, and to develop the database (including a prototype for proof of concept).

The data are to be collected by the contractor and/or IV&V agent and input into the database by the PM's Office. In cases where the developer does the collection, it is recommended that the IV&V agent should at least sample some of the more crucial metric data elements for consistency and correctness. The PM would only have access to their project. Program Executive Officers (PEOs) would only have access to data on the programs under their purview. Decision makers, testers, assessors, and evaluators would have access to data on all projects.

13. RECOMMENDATIONS

In addition to the many recommendations specified in the description of each metric's use, interpretation, and rules of thumb, the following recommendations are made in order to improve the state of the Army in software development, test, and evaluation.

The metrics must be implemented by force. That is, there must be a regulatory requirement throughout the Army, and must be used for all systems (both information systems and tactical/embedded systems) undergoing development or a major post-deployment upgrade. The most important requirement, however, is that the data to be collected must be specified in the RFP and scope of work (SOW). The contractual language required to obtain these data needs to be developed.

The Army should continue to monitor (and increase its participation in) the efforts of the various metric-related working groups being sponsored by the SEI. These groups consist of nationally recognized experts, and include many of the key researchers. It is hoped that their efforts, as they near fruition, can be used to improve the Army's recommended set of metrics contained in this report.

The Army should make sure this metric definition effort dovetails with the developing DOD Software Action Plan. The efforts of the Software Action Plan and the STEP should be complimentary efforts, especially in the critical area of metric development. To date, there has been some initial coordination between the two groups.

The Army should avail itself of any opportunities for tri-service coordination on metrics. It is recognized that the Air Force has been a leader in many aspects of software evaluation. It is also known that the Air Force Systems Command is leading an effort to develop a similar set of metrics as is specified in this report. The Army should monitor this effort. One of the most important reasons to consider tri-service coordination is that if all the services were to collect the same data elements, then the validation process and stepwise refinement of the metric set could be accomplished much sooner.

The Army-wide data base described in Section 12 should be completed to track the use of metrics. Such a data base would serve as a repository of the use of metrics (including cost data), and would be the focal point for efforts to refine and formally validate the metric set. Also, the existence of such a

data base would provide the opportunity for the conduct of cost benefit analyses.

Finally, the metrics must be used as an entire set. Within the use and interpretation description of each metric, we have proposed specific ways in which to look at several metrics together, so as to get a more complete picture of the state of the software. Also, several metrics have somewhat limited life cycle applicability, and various metrics are more meaningful at certain times in the life cycle. Many individuals might also express reservations about the possible "gaming" of metric data by the developer or contractor. Using the metrics as an entire set can serve to mitigate these concerns and risks.

14. CONCERNS

Any serious known limitations for a particular metric were described in the use and interpretation section for that particular metric. Additional concerns are specified in the following paragraphs.

One of the primary concerns of the group is that software engineering as not a mature process. It is widely recognized that the discipline of software engineering is still evolving, as evidenced by the continual introduction of new technologies and software development techniques. Within private industry, there is inconsistent application of recognized software development methodologies and requisite metrics to measure those processes and products. Within the Army, there does not seem to be discipline with either the use of software engineering principles or the collection or evaluation of metric data. The hope of the authors is that the process of measuring software can help to bring the disciplines of engineering to the process, so that the process can indeed be referred to as "software engineering."

The concern about the immaturity of software engineering leads to the next concern. The SEI, which is recognized and chartered as the DOD's leader to advance the state of software engineering and transition key software technologies to the DOD, feels that it is too early to definitize "the" set of metrics. SEI's planned experimentation is aimed at studying the motivation of the software practitioner, so that the most important factors to measure can be identified. We are concerned about recommending a set of metrics in the face of SEI's concern. However, we recognize the need for continued study and an iterative process of metric refinement and eventual validation. The stepwise refinement process should include any findings of the SEI. We feel strongly that it is important to start measuring the software development process and associated products. Just to measure by itself should add value because people will know that they are being monitored. Process improvements cannot come about until measurements are made of that process and its products.

Several metrics are closely tied to the current generation of languages (third generation, procedurally-oriented languages) and environments typically employed by the Army. As mentioned earlier, while some language in the description of the metrics is tied to the current life cycle, the so-called "waterfall" model specified in DOD-STD-2167A, the Measures Subgroup holds the

firm belief that the metrics can and should be applied to systems using other life cycle models, including the spiral model, and non-traditional development techniques such as rapid prototyping and evolutionary development. As we move toward applications which use fifth generation languages, computer aided software engineering (CASE), object-oriented programming, and a wide range of other techniques, the current set of metrics will have to be improved.

It must be pointed out that this recommended, minimum set of metrics may not be "the set." However, as we have hopefully pointed out throughout this report, we feel that it represents a logical and good starting point for departure on a software measurement effort within the Army.

Finally, even if the validation process proves out the metric set as a truly predictive and valuable tool for program success, many other key activities beyond the implementation of a metrics program still have to be performed to guarantee the success of a development effort. Besides the fact that the metric set is not a silver bullet, it is also recognized that, in accordance with the spirit of TQM, other process improvements need to be continuously pursued. As can be readily seen in Table 12-1, while the metrics do provide considerable support for some quality factors, they are not sufficient by themselves to ensure a quality program and product.

APPENDIX A - OPTIONAL METRICS

APPENDIX A - OPTIONAL METRICS

A.1 Metric: Manpower

Purpose/Description:

This measure provides an indication of the developer's application of human resources to the developmental program and to maintain sufficient staffing for completion of the project. It can also provide indications of possible problems with meeting schedule and budget. It is used to examine the various elements involved in staffing a software project. These elements include the planned level of effort, the actual level of effort, and the losses in the software staff measured per labor category. Planned manpower profiles are derived from the appropriate planning documents submitted to the government. These are usually provided in the developer's proposal or the Software Development Plan. The planned level of effort is the number of labor hours that are scheduled to be worked on a CSCI each month. The planned levels are compared with the actual over a given time period to give the government insight into the deviations between them. These deviations can be monitored to ensure that the developer is meeting the necessary staffing criteria.

Life Cycle Application:

Track for entire length of development (including PDSS).

Algorithm/Graphical Display:

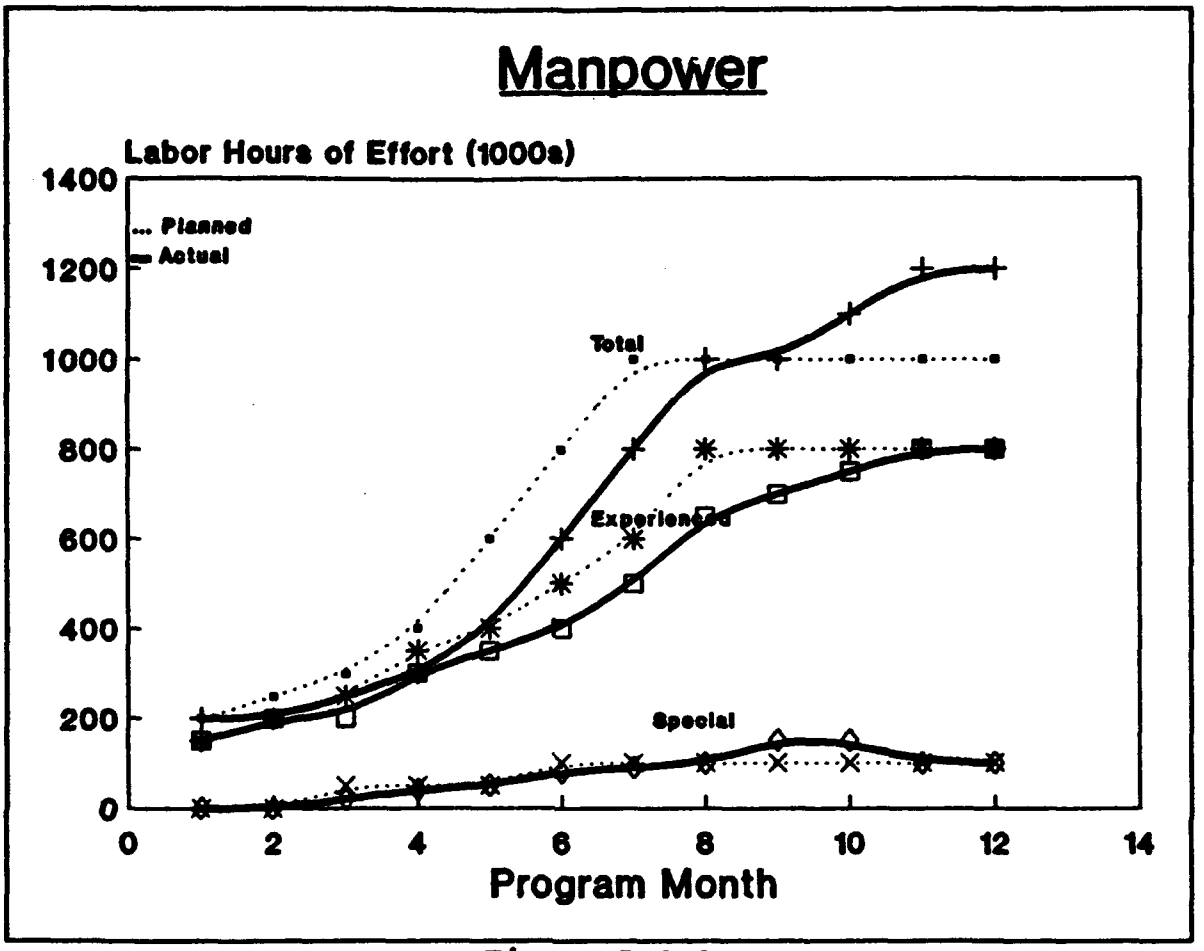

Figure A.1-1

Data Requirements:

Notes:

1. Special skills personnel are those individuals who possess specialized software-related abilities defined as crucial to the success of the particular system. For example, Ada programmers or artificial intelligence experts might be considered special skills personnel for one project, but not necessarily for another project.
2. Experienced personnel are defined as those individuals with a minimum of three years experience in software development for similar applications.
3. Total personnel are the sum of experienced and inexperienced personnel. Special skills personnel are counted within the broad categories of experienced and inexperienced, but are also tracked separately.
4. Other categories can certainly be created for aspects of a program which are deemed worthy of special attention. For many projects, software quality assurance people might be tracked as a separate category.

for each personnel type (total, experienced, special) :
labor hours expended each month
number people lost
number people gained
labor hours planned each of month

Frequency of Reporting:

monthly

Use/Interpretation:

The software staff includes, as a minimum, those engineering and management personnel directly involved with software system planning, requirements definition, design, coding, integration, test, documentation, configuration management, and quality assurance. Losses and gains for each category specified above should be tracked monthly to indicate potential problem areas. Personnel who have been replaced are still counted as a loss. High turnover of experienced personnel can adversely affect project success. Also, for example, adding many personnel (beyond those numbers planned) late in the development process may provide an indication of impending problems. Turnover of key people must also be watched closely.

Significant deviations from planned levels can be used as an indicator of potential problems with staffing the various software development activities. Deviations between actual levels and planned can be detected, explained and corrected before they negatively impact the development schedule. The losses in the staff can be monitored to detect a growing trend or significant loss of experienced staff. This indicator assists the government in determining if the developer has scheduled a sufficient number of employees to produce the product in the time allotted.

The shape of the staff profile trend curve tends to start at a moderate level at the beginning of the contract, grow through design, peak at coding/testing and diminish near the completion of integration testing. Individual labor categories, however, are likely to peak at different points in the life cycle. The optimum result would show little deviation between the planned and actual levels for each category with losses kept to a minimum. Specific attention should be paid to any case where there is a significant deviation (+/- 10%) between actual and planned. In the case where actual is more than planned, this may suggest that the developer:

- underestimated the work involved
- found out that the task was more complicated than expected
- did not perform the work efficiently
- is ahead of schedule
- is behind schedule and is adding manpower to catch up
- is adding people to make up for a lack of experienced ones

If the actual levels are less than planned, this may suggest that the developer:

- overestimated the work involved
- did not perform the task completely
- the effort was not as complex as expected
- performed the work efficiently
- did not assign adequate manpower to the task
- misinterpreted the task or requirements
- is ahead of schedule

In cases of large deviation the developer should be required to determine the cause and report any corrective actions necessary.

The manpower metrics are used primarily for project management and do not necessarily have a direct relationship with other technical and maturity metrics. For example, manpower levels are usually higher during testing activities. This does not necessarily reflect an increase in the quality levels of the product or suggest that the depth of testing metrics will be higher.

The manpower metrics should be used in conjunction with the development progress and test coverage metrics.

The value of this metric is somewhat tied to the accuracy of the development and staffing plan, as well as to the accuracy of the labor reporting system.

Rules of Thumb:

A high ratio of total to experienced personnel is undesirable. A ratio of 3:1 is typical.

Significant deviations from the planned staffing profile, as well as a high turnover rate in any category, should be investigated so as to minimize risk to the government.

When the developer has expended 80% of their planned or budgeted resources, it shall be identified and highlighted to the government and be documented in the monthly reports. Closer attention should then be paid to the remaining resources.

References:

"Software Reporting Metrics", The Mitre Corporation, ESD-TR-85-145, MTR 9650 Revision 2, November 1985.

"Software Management Indicators", Air Force Systems Command, AFSCP 800-43, January 31, 1986.

"Software Management Indicators, Management Insight",AMC-P 70-13, 31 January 1987.

"Revised Implementation Guidelines for Software Management and Quality Indicators for Advanced Field Artillery Tactical Data System", 30 Jul 89.

A.2 Metric: Development Progress

Purpose/Description:

The development progress metrics provide indications of the degree of completeness of the software development effort, and hence can be used to judge readiness to proceed to the next stage of software development.

Life Cycle Application:

Begin collecting at PDR and continue for the entire software development phase.

Algorithm/Graphical Display:

(Note: the following calculations can be performed at either the CSC, CSCI, or system level.)

Compute percent of CSUs 100% designed.
Compute percent of CSUs 100% coded and successfully unit tested.
Compute percent of CSUs 100% integrated.

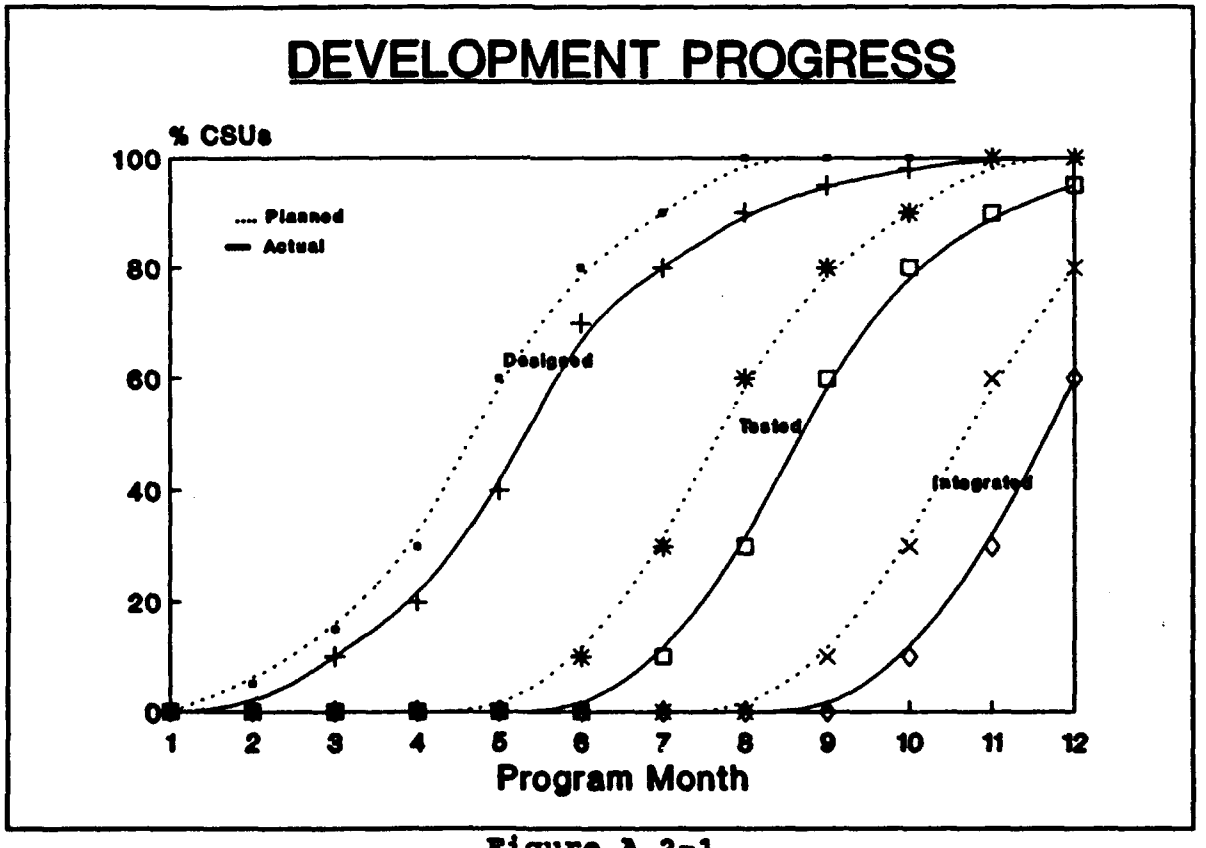

Figure A.2-1

Additionally, by use of the requirements traceability matrix, one can plot functionality (which has been developed and verified) versus time as a measure of development progress.

Data Requirements:

CSCI/CSC/CSU development, test and integration schedules
number of CSUs per CSCI
number of CSUs 100% designed and reviewed by government
number of CSUs 100% coded and successfully unit tested
number of CSUs 100% integrated into a CSC or CSCI

NOTE: "Successfully" tested is defined as completing all test cases (required test coverage or depth) with no defects. "Integrated" is defined as being actually and logically connected (in a static sense) with all required modules. (Dynamic tasking is not considered here).

Frequency of Reporting:

monthly

Use/Interpretation:

The design, coding, unit testing, and integration of CSUs should progress at a reasonable rate. Plotting the progress in these three categories versus what was originally planned can give indications of potential problems with schedule and cost. In certain instances, consideration must be given to a possible re-baselining of the software (e.g., in an evolutionary approach) or if one simply must add modules due to changes in the requirements.

The development progress metrics should be used with the test coverage metrics (breadth and depth of testing) to assess the readiness to proceed to a formal government test. They should also be used with the requirements traceability metrics so that progress can be tracked in consonance with the tracing of requirements. Additionally, it can be used with the schedule metric to help evaluate schedule risk.

The development progress metrics should be used with the manpower metrics to identify areas where the developer is experiencing problems. Also, using these metrics with the computer resource utilization metrics can ensure that the actual utilization is representative of a complete system. Finally, special attention should be given to the development progress of high complexity CSUs.

These metrics pass no judgement on the achievability of the contractor's development plan.

Rules of Thumb:

One hundred percent (100%) of all CSUs should be designed prior to proceeding beyond CDR for the appropriate CSCI.

One hundred percent (100%) of all CSUs should be coded, successfully tested, and integrated before proceeding to a formal system level government test.

References:

"Software Reporting Metrics", The Mitre Corporation, ESD-TR-85-145, MTR 9650 Revision 2, November 1985.

"Software Management Indicators", Air Force Systems Command, AFSCP 800-43, January 31, 1986.

ACRONYM LIST

Acronym	Definition
ACAT	Acquisition Category
ACWP	Actual Cost of Work Performed
AFOTEC	Air Force Operational Test and Evaluation Center
AIS	Automated Information System
AMC	Army Materiel Command
AMCCOM	Armament, Munitions, and Chemical Command
AMSAA	Army Materiel Systems Analysis Activity
ASAP	Army Streamlined Acquisition Program
ASARC	Army Systems Acquisition Review Council
ATCCS	Army Tactical Command and Control System
BCWP	Budgeted Cost of Work Performed
BCWS	Budgeted Cost of Work Scheduled
CASE	Computer Aided Software Engineering
CDR	Critical Design Review
CECOM	Communications and Electronics Command
CFSR	Contract Funds Status Report
CPU	Central Processing Unit
CRU	Computer Resource Utilization
CRWG	Computer Resource Working Group
CSC	Computer Software Component
C/SCSC	Cost/Schedule Control System Criteria
CSCI	Computer Software Configuration Item
C/SSR	Cost/Schedule Status Report
CSU	Computer Software Unit
CWBS	Contract Work Breakdown Structure
DA	Department of the Army
DAB	Defense Acquisition Board
DID	Data Item Description
DOD	Department of Defense
DP	Design Progress
EAC	Estimated Cost At Completion
ECP-S	Engineering Change Proposal - Software
FCA	Functional Configuration Audit
FQT	Formal Qualification Test
HDBK	Handbook
HW	Hardware
I/O	Input / Output
IOTE	Independent Operational Test and Evaluation
IPR	In Process Review
IRS	Interface Requirements Specification
ISEC	Information Systems Engineering Command
ISSC	Information Systems Support Command

Acronym	Definition
LAN	Local Area Network
LCSSC	Life Cycle Software Support Center
LOC	Line of Code
MAISRC	Major Automated Information Systems Review Council
MCCR	Mission Critical Computer Resources
MR	Management Reserve
MS	Milestone
MSCR	Materiel Systems Computer Resources
MTB SW MF	Mean Time Between Software Mission Failure
MTB HW/SW MF	Mean Time Between Hardware/Software Mission Failure
MTBF	Mean Time Between Failure
MTBOMF	Mean Time Between Operational Mission Failure
OMS/MP	Operational Mode Summary / Mission Profile
OPTEC	Operation Test and Evaluation Command
ORD	Operational Requirements Document
OT	Operational Test
PCA	Physical Configuration Audit
PDL	Program Design Language
PDR	Preliminary Design Review
PDSS	Post Deployment Software Support
PEO	Program Executive Officer
PM	Program Manager
PPQT	Pre-Production Qualification Test
RADC	Rome Air Development Center
RAM	Random Access Memory
REQTS	Requirements
RFP	Request For Proposal
SDD	Software Design Document
SDP	Software Development Plan
SDR	System Design Review
SEI	Software Engineering Institute
SOW	Scope Of Work
SPS	Software Product Specification
SRR	System Requirements Review
SRS	Software Requirements Specification
SRTM	Software Requirements Traceability Matrix
SS	System Specification
SSR	Software Specification Review
SSS	System / Segment Specification
STEP	Software Test and Evaluation Panel
STD	Software Test Description
STP	Software Test Plan
STR	Software Trouble Report
SW or S/W	Software

Acronym	Definition
TECOM	Test and Evaluation Command
TIWG	Test Integration Working Group
TQM	Total Quality Management
TT	Technical Test
UFD	Users' Functional Description
VDD	Version Description Document
VS	Versus
WBS	Work Breakdown Structure

DISTRIBUTION LIST

No. of Copies	Organization
1	Commander U.S. Army Materiel Command ATTN: AMCCE-QE 5001 Eisenhower Avenue Alexandria, VA 22302
2	Commander U.S. Army Operational Evaluation Command ATTN: CSTE-ZT, CSTE-ESE-S Park Center IV 4501 Ford Avenue Alexandria, VA 22302-1458
1	Office of the Deputy Under Secretary of the Army for Operations Research ATTN: SAUS-OR The Pentagon Washington, DC 20310-0102
2	Software Engineering Institute Carnegie Mellon University ATTN: Anita Carleton, Robert Park Pittsburgh, PA 15213
1	Director U.S. Army Test and Evaluation Management Agency ATTN: DACS-TE (Dr. Foulkes) The Pentagon Washington, DC 20310-0102
1	Director of Information Systems for Command, Control, Communications, and Computers Headquarters, Department of the Army ATTN: SAIS-ADW The Pentagon Washington, DC 20310-0107

DISTRIBUTION LIST

No. of Copies	Organization
	Aberdeen Proving Ground
2	Commander U.S. Army Test and Evaluation Command ATTN: AMSTE-TA, AMSTE-IS-P Aberdeen Proving Ground, MD 21005
45	Director U.S. Army Materiel Systems Analysis Activity ATTN: AMXSY-A (5 cys) AMXSY-G (5 cys) AMXSY-C (7 cys) AMXSY-CA (12 cys) AMXSY-R (6 cys) AMXSY-RE (2 cys) AMXSY-L (6 cys) AMXSY-DA (2 cys) Aberdeen Proving Ground, MD 21005-5071

AMXSY-CA

Study Gist

Subject : AMSAA Technical Report Number TR-532, "Army Software Test and Evaluation Panel (STEP) Software Metrics Initiatives Report"

Principal Findings : The primary finding is a set of twelve software metrics for use in measuring and evaluating the state of software in Army development programs. The set includes metrics covering quality, requirements, and management. Each metric is described in terms of a purpose / description, algorithm, data requirements, frequency of reporting, use / interpretation, and rules of thumb. Recommendations are also made concerning use of the metrics in various phases of the software and system life cycle.

Main Assumptions : More discipline is needed in managing Army software development. These metrics provide insight into the process and product in such a way that progress is demonstrated before proceeding to the next phase of development or testing.

Principal Limitations : The twelve metrics have not been formally validated, although each metric has been used successfully by some segments of the military software development community. Further, the set of metrics represents a logical starting point for the Army metrics program.

Scope of the Effort : The study involved a group of Army software experts, including developers, testers, and evaluators. The effort spanned about two years from initial meetings to creation of the final report.

Objective : The objective was to develop a minimum set of software metrics, spanning process and product measures, for mandatory use on all Army software development programs.

Basic Approach : Government, industry, and academic experts were consulted. Also, a literature review was conducted for software metrics research and applications. Finally, sample Army development programs were reviewed for metrics usage.

Reason for Performing the Study or Analysis : The reason for the study was to support the Army Software Test and Evaluation Panel (STEP) attempt to bring more discipline to Army software development efforts.

Impact of the Study : The recommended set of twelve software metrics has been incorporated into DA Pamphlet 73-1, Part 7, "Software Test and Evaluation Guidelines."

Sponsor : The Deputy Under Secretary of The Army for Operations Research (DUSA-OR).

Principal Investigators : Patrick J. O'Neill and Henry P. Betz

Name/Address/Phone Number Where Comments & Questions Can Be Sent :
U.S. Army Materiel Systems Analysis Activity
ATTN: AMXSY-CA (Mr. O'Neill)
Aberdeen Proving Ground, MD 21005-5071
(410) 278-6429

Defense Technical Information Center (DTIC) Accession Number of Final Report :

Other Than Sponsor, Who Could Benefit From This Study/Information :
Anyone interested in bring more discipline to the software development, test, and evaluation process.